

A Strong Distillery

Beniamino Accattoli¹, Pablo Barenbaum², and Damiano Mazza³

¹ INRIA, UMR 7161, LIX, École Polytechnique, CNRS
beniamino.accattoli@inria.fr

² University of Buenos Aires – CONICET
pbarenbaum@dc.uba.ar

³ CNRS, UMR 7030, LIPN, Université Paris 13, Sorbonne Paris Cité
Damiano.Mazza@lipn.univ-paris13.fr

Abstract. Abstract machines for the strong evaluation of λ -terms (that is, under abstractions) are a mostly neglected topic, despite their use in the implementation of proof assistants and higher-order logic programming languages. This paper introduces a machine for the simplest form of strong evaluation, leftmost-outermost (call-by-name) evaluation to normal form, proving it correct, complete, and bounding its overhead. Such a machine, deemed *Strong Milner Abstract Machine*, is a variant of the KAM computing normal forms and using just one global environment. Its properties are studied via a special form of decoding, called a *distillation*, into the Linear Substitution Calculus, neatly reformulating the machine as a standard micro-step strategy for explicit substitutions, namely *linear leftmost-outermost reduction*, *i.e.* the extension to normal form of linear head reduction. Additionally, the overhead of the machine is shown to be linear both in the number of steps and in the size of the initial term, validating its design. The study highlights two distinguished features of strong machines, namely backtracking phases and their interactions with abstractions and environments.

1 Introduction

The computational model behind functional programming is the weak l -calculus, where *weakness* is the fact that evaluation stops as soon as an abstraction is obtained. Evaluation is usually defined in a small-step way, specifying a strategy for the selection of weak β -redexes. Both the advantage and the drawback of l -calculus is the lack of a machine in the definition of the model. Unsurprisingly implementations of functional languages have been explored for decades.

Implementation schemes are called *abstract machines*, and usually account for two tasks. First, they switch from small-step to *micro-step* evaluation, delaying the costly meta-level substitution used in small-step operational semantics and replacing it with substitutions of one occurrence at a time, when required. Second, they also *search the next redex* to reduce, walking through the program according to some evaluation strategy. Abstract machines are *machines* because they are deterministic and the complexity of their steps can easily be measured,

and are *abstract* because they omit many details of a real implementation, like the actual representation of terms and data-structures or the garbage collector.

Historically, the theory of *l*-calculus and the implementation of functional languages have followed orthogonal approaches. The former rather dealt with *strong* evaluation, and it is only since the seminal work of Abramsky and Ong [1] that the theory took weak evaluation seriously. Dually, practical studies mostly ignored *strong* evaluation, with the notable exception of Crégut [12,13] (1990) and, more recently, the semi-strong approach of Grégoire and Leroy [22] (2002)—see also the *related work* paragraph below. Strong evaluation is nonetheless essential in the implementation of proof assistants or higher-order logic programming, typically for type-checking in frameworks with dependent types as the Edinburgh Logical Framework or the Calculus of Constructions, as well as for unification modulo $\beta\eta$ in simply typed frameworks like *l-prolog*.

The aim of this paper is to move the first steps towards a systematic and theoretical exploration of the implementation of strong evaluation. Here we deal with the simplest possible case, call-by-name evaluation to strong normal form, implemented by a variant of the Krivine Abstract Machine. The study is carried out according to the *distillation methodology*, a new approach recently introduced by the authors and previously applied only to weak evaluation [3].

Distilling Abstract Machines. Many abstract machines can be rephrased as strategies in *l-calculi with explicit substitutions* (ES for short), see at least [14,23,13,9,24,8]. The Linear Substitution Calculus (LSC)—a variation over a *l*-calculus with ES by Robin Milner [26] developed by Accattoli and Kesner [2,4]—provides more than a simple reformulation: it disentangles the two tasks carried out by abstract machines, retaining the *micro-step operational semantics* and omitting the *search for the next redex*. Such a neat disentangling, that we prefer to call a *distillation*, is a decoding based on the following key points:

1. *Partitioning*: the machine transitions are split in two classes. *Principal transitions* are mapped to the rewriting rules of the calculus, while *commutative transitions*—responsible for the search for the redex—are mapped on a notion of structural equivalence, specific to the LSC.
2. *Rewriting*: structural equivalence accounts both for the search for the redex and garbage collection, and commutes with evaluation. It can thus be postponed, isolating the micro-step strategy in the rewriting of the LSC.
3. *Logic*: the LSC itself has only two rules, corresponding to cut-elimination in linear logic proof nets. A distillation then provides a logical reading of an abstract machine (see [3] for more details).
4. *Complexity*: by design, a principal transition has to take linear time in the input, while a commutative transition has to be constant.

A *distillery* is then given by a machine, a strategy, a structural equivalence, and a decoding function satisfying the above points. In bilinear distilleries, the number of commutative transitions is linear in both the *number of principal transitions* and the *size of the initial term*. Bilinearity guarantees that distilling away the commutative part by switching to the LSC preserves the asymptotical

behavior, *i.e.* it does not forget too much. At the same time, the bound on the commutative overhead justifies the design of the abstract machine, providing a provably bounded implementation scheme.

A Strong Distillery. Our machine is a strong version of the Milner Abstract Machine (MAM), a variant with just one *global environment* of the Krivine Abstract Machine (KAM), introduced in [3].

The first result of the paper is the design of a distillery relating the Strong MAM to *linear leftmost-outermost reduction* in the LSC [4,5]—that is at the same time a refinement of leftmost-outermost (LO) β -reduction and an extension of linear head reduction [25,15,2] to normal form—together with the proof of correctness and completeness of the implementation [27]. Moreover, the linear LO strategy is *standard* and *normalizing* [4], and thus we provide an instance of Plotkin’s approach of mapping abstract machines to such strategies [4].

The second result is the complexity analysis showing that the distillery is bilinear, *i.e.* that the cost of the additional search for the next redex specific to the machine is negligible. The analysis is simple, and yet subtle and robust. It is subtle because it requires a global analysis of executions, and it is robust because the overhead is bilinear for *any* evaluation sequence, not necessarily to normal form, and even for diverging ones.

For the design of the Strong MAM we make various choices:

1. *Global Environment*: we employ a *global* environment, which is in opposition to having closures (pairing subterms with *local* environments), and it models a store-based implementation scheme. The choice is motivated by future extensions to more efficient strategies as call-by-need, where the global environment allows to integrate sharing with a form of memoization [17,3].
2. *Sequential Exploration and Backtracking*: we fix a sequential exploration of the term (according to the leftmost-outermost order), in opposition to the parallel evaluation of the arguments (once a head normal form has been reached). This choice internalizes the handling of the recursive iterations, that would be otherwise left to the meta-level, providing a finer study of the data-structures needed by a strong machine. On the other hand, it forces to have backtracking transitions, activated when the current subterm has been checked to be normal and evaluation needs to retrieve the next subterm on the stack. Call-by-value machines usually have a similar but simpler backtracking mechanism, realized via an additional component, the *dump*.
3. *(Almost) No Garbage Collection*: we focus on time complexity, and thus ignore space issues, that is, our machine does not account for garbage collection. In particular, we keep the global environment completely unstructured, similarly to the (weak) MAM. Strong evaluation however is subtler, as to establish a precise relationship between the machine and the calculus with ES, garbage collection cannot be completely ignored. Our approach is to isolate it within the meta-level: we use a system of parenthesized markers, to delimit subenvironments created under abstractions that could be garbage collected once the machine backtracks outside those abstraction. These labels are not inspected by the transitions, and play a role only for the proof of

the distillation theorem. Garbage collection then is somewhat accounted for by the analysis, but there are no dedicated transitions nor rewriting rules, it is rather encapsulated in the decoding and in the structural equivalence.

Efficiency? It is known that LO evaluation is not efficient. Improvements are possible along three axis: refining the strategy (by turning to strong call-by-value/need, partially done in [22,13,7]), speeding up the substitution process (by forbidding the substitution of variables, see [6,7]), and avoiding useless substitutions (by adding *useful sharing*, see [5,7]). These improvements however require sophisticated machines, left to future work.

LO evaluation is nonetheless a good first case study, as it allows to isolate the analysis of backtracking phases and their subtle interactions with abstractions and environments. We expect that the mentioned optimizations can be added in a quite modular way, as they have all been addressed in the complementary study in [7], based on the same technology (*i.e.* LSC and distilleries).

(Scarce) Related Work. Beyond Crégut's [12,13], we are aware of only two other similar works on strong abstract machines, García-Pérez, Nogueira and Moreno-Navarro's [21] (2013), and Smith's [29] (unpublished, 2014). Two further studies, de Carvalho's [11] and Ehrhard and Regnier's [19], introduce strong versions of the KAM but for theoretical purposes; in particular, their design choices are not tuned towards implementations (*e.g.* rely on a naïve parallel exploration of the term). Semi-strong machines for call-by-value (*i.e.* dealing with weak evaluation but on open terms) are studied by Grégoire and Leroy [22] and in a recent work by Accattoli and Sacerdoti Coen [7] (see [7] for a comparison with [22]). More recent work by Dénès [18] and Boutiller [10] appeared in the context of term evaluation in Coq. These works, which do offer the nice perspective of concretely dealing with proof assistants, are focused on quite specific Coq-related tasks (such as term simplification) and the difference in reduction strategy and underlying motivations makes a comparison difficult.

Of all the above, the closest to ours is Crégut's work, because it defines an implementation-oriented strong KAM, thus also addressing leftmost-outermost reduction. His machine uses local environments, sequential exploration and backtracking, scope markers akin to ours, and a calculus with ES to establish the correctness of the implementation. His calculus, however, has no less than 13 rewriting rules, while ours just 2, and so our approach is simpler by an order of magnitude. Moreover, we want to stress that our contribution does not lie in the machine *per se*, or the chosen reduction strategy (as long as it is strong), but in the combined presence of a robust and simple abstraction of the machine, provided by the LSC, and the complexity analysis showing that such an abstraction does not miss too much. In this respect, none of the above works comes with an analysis of the overhead of the machine nor with the logical and rewriting perspective we provide. In fact, our approach offers general guidelines for the design of (strong) abstract machines. The choice of leftmost-outermost reduction showcases the idea while keeping technicalities to a minimum, but it is by no means a limitation. The development of strong distilleries for call-by-value

or lazy strategies, which may be more attractive from a programming languages perspective, are certainly possible and will be the object of future work (again, an intermediary step has already been taken in [7]).

Global environments are explored by Fernández and Siafakas in [20], and used in a minority of works, *e.g.* [28,17]. We introduced the distillation technique in [3] to revisit the relationship between the KAM and weak linear head reduction pointed out by Danos and Regnier [15]. Distilleries have also been used in [7]. The idea to distinguish between *operational content* and *search for the redex* in an abstract machine is not new, as it underlies in particular the *refocusing semantics* of Danvy and Nielsen [16]. The LSC, with its roots in linear logic proof nets, allows to see this distinction as an avatar of the principal/commutative divide in cut-elimination, because machine transitions may be seen as cut-elimination steps [8,3]. Hence, it is fair to say that distilleries bring an original refinement where logic, rewriting, and complexity enlighten the picture, leading to formal bounds on machine overheads.

Omitted proofs may be found in the appendices.

2 Linear Leftmost-Outermost Reduction

The language of the *linear substitution calculus* (LSC for short) is given by the following term grammar:

$$\text{LSC Terms} \quad t, u, w, r ::= x \mid lx.t \mid tu \mid t[x \leftarrow u].$$

The constructor $t[x \leftarrow u]$ is called an *explicit substitution, shortened ES* (of u for x in t). Both $lx.t$ and $t[x \leftarrow u]$ bind x in t , and we silently work modulo α -equivalence of these bound variables, *e.g.* $(xy)[y \leftarrow t]\{x \leftarrow y\} = (yz)[z \leftarrow t]$.

The operational semantics of the LSC is parametric in a notion of (one-hole) context. General *contexts*, that simply extend the contexts for *l*-terms with the two cases for ES, and the special case of *substitution contexts* are defined by:

$$\begin{array}{ll} \text{Contexts} & C, C' ::= \langle \cdot \rangle \mid lx.C \mid Ct \mid tC \mid C[x \leftarrow t] \mid t[x \leftarrow C]; \\ \text{Substitution Contexts} & L, L' ::= \langle \cdot \rangle \mid L[x \leftarrow t]. \end{array}$$

We write $C \prec_p t$ if there is a term u s.t. $C\langle u \rangle = t$, call it the *prefix relation*.

The rewriting relation is $\rightarrow := \rightarrow_m \cup \rightarrow_e$ where \rightarrow_m and \rightarrow_e are the *multiplicative* and *exponential* rules, defined by

	RULE AT TOP LEVEL	CONTEXTUAL CLOSURE
Multiplicative	$L\langle lx.t \rangle u \mapsto_m L\langle t[x \leftarrow u] \rangle$	$C\langle t \rangle \rightarrow_m C\langle u \rangle$ if $t \mapsto_m u$
Exponential	$C\langle x \rangle [x \leftarrow u] \mapsto_e C\langle u \rangle [x \leftarrow u]$	$C\langle t \rangle \rightarrow_e C\langle u \rangle$ if $t \mapsto_e u$

The rewriting rules are assumed to use *on-the-fly* α -equivalence to avoid variable capture. For instance, $(\lambda x.t)[y \leftarrow u]y \rightarrow_m t[y \leftarrow z][x \leftarrow y][z \leftarrow u]$ for $z \notin \text{fv}(t)$, and $(\lambda y.(xy))[x \leftarrow y] \rightarrow_e (\lambda z.(yz))[x \leftarrow y]$. Moreover, in \rightarrow_e the context C is assumed to not capture x , in order to have $(lx.x)[x \leftarrow y] \not\rightarrow_e (lx.y)[x \leftarrow y]$.

The above operational semantics ignores garbage collection. In the LSC, this may be realized by an additional rule which may always be postponed, see [2].

Taking the external context into account, an exponential step has the form $C' \langle C \langle x \rangle [x \leftarrow u] \rangle \rightarrow_e C' \langle C \langle u \rangle [x \leftarrow u] \rangle$. We shall often use a *compact* form:

$$\begin{array}{c} \text{EXPONENTIAL RULE IN COMPACT FORM} \\ C'' \langle x \rangle \rightarrow_e C'' \langle u \rangle \quad \text{if } C'' = C' \langle C[x \leftarrow u] \rangle \end{array}$$

Definition 1 (Redex Position). Given a \rightarrow_m -step $C \langle t \rangle \rightarrow_m C \langle u \rangle$ with $t \mapsto_m u$ or a compact \rightarrow_e -step $C \langle x \rangle \rightarrow_e C \langle t \rangle$, the position of the redex is the context C .

We identify a redex with its position, thus using C, C', C'' for redexes, and use $d : t \rightarrow^k u$ for derivations, *i.e.* for possibly empty sequences of rewriting steps. We write $|t|_{[.]}$ for the number of substitutions in t , and use $|d|$, $|d|_m$, and $|d|_e$ for the number of steps, m -steps, and e -steps in d , respectively.

Linear Leftmost-Outermost Reduction, Two Definitions. We give two definitions of linear LO reduction \rightarrow_{LO} , a traditional one based on ordering redexes and a new contextual one not mentioning the order, apt to work with LSC and relate it to abstract machines. We start by defining the LO order on contexts.

Definition 2 (LO Order). The outside-in order $C \prec_O C'$ is defined by

1. Root: $\langle \cdot \rangle \prec_O C$ for every context $C \neq \langle \cdot \rangle$;
2. Contextual closure: if $C \prec_O C'$ then $C'' \langle C \rangle \prec_O C'' \langle C' \rangle$ for any context C'' .

Note that \prec_O can be seen as the prefix relation \prec_p on contexts. The left-to-right order $C \prec_L C'$ is defined by

1. Application: if $C \prec_p t$ and $C' \prec_p u$ then $Cu \prec_L tC'$;
2. Substitution: if $C \prec_p t$ and $C' \prec_p u$ then $C[x \leftarrow u] \prec_L t[x \leftarrow C']$;
3. Contextual closure: if $C \prec_L C'$ then $C'' \langle C \rangle \prec_L C'' \langle C' \rangle$ for any context C'' .

Last, the left-to-right outside-in order is defined by $C \prec_{LO} C'$ if $C \prec_O C'$ or $C \prec_L C'$.

Two examples of the outside-in order are $(lx.\langle \cdot \rangle)t \prec_O (lx.(\langle \cdot \rangle[y \leftarrow u]))t$ and $t[x \leftarrow \langle \cdot \rangle] \prec_O t[x \leftarrow uC]$, and an example of the left-to-right order is $t[x \leftarrow C]u \prec_L t[x \leftarrow w]\langle \cdot \rangle$. The next immediate lemma guarantees that we defined a total order.

Lemma 1 (Totality of \prec_{LO}). If $C \prec_p t$ and $C' \prec_p t$ then either $C \prec_{LO} C'$ or $C' \prec_{LO} C$ or $C = C'$.

Remember that we identify redexes with their position context and write $C \prec_{LO} C'$. We can now define linear LO reduction, first considered in [4], where it is proved that it is standard and normalizing, and then in [5], extending linear head reduction [25,15,2] to normal form.

Definition 3 (Linear LO Reduction \rightarrow_{LO}). Let t be a term. C is the leftmost-outermost (LO for short) redex of t if $C \prec_{\text{LO}} C'$ for every other redex C' of t . We write $t \rightarrow_{\text{LO}} u$ if a step reduces the LO redex.

We now define LO contexts and prove that the position of a linear LO step is always a LO context. We need two notions.

Definition 4 (Neutral Term). A term is neutral if it is \rightarrow -normal and it is not of the form $L(\lambda x.t)$.

Neutral terms are such that their plugging in a context cannot create a multiplicative redex. We also need the notion of left free variable of a context, i.e. of a variable occurring free at the left of the hole.

Definition 5 (Left Free Variables). The set $\text{lfv}(C)$ of left free variables of C is defined by:

$$\begin{array}{ll} \text{lfv}(\langle \cdot \rangle) := \emptyset & \text{lfv}(tC) := \text{fv}(t) \cup \text{lfv}(C) \\ \text{lfv}(lx.C) := \text{lfv}(C) \setminus \{x\} & \text{lfv}(C[x \leftarrow t]) := \text{lfv}(C) \setminus \{x\} \\ \text{lfv}(Ct) := \text{lfv}(C) & \text{lfv}(t[x \leftarrow C]) := (\text{fv}(t) \setminus \{x\}) \cup \text{lfv}(C) \end{array}$$

Definition 6 (LO Contexts). A context C is LO if

1. Right Application: whenever $C = C' \langle tC'' \rangle$ then t is neutral, and
2. Left Application: whenever $C = C' \langle C''t \rangle$ then $C'' \neq L(\lambda x.C''')$.
3. Substitution: whenever $C = C' \langle C''[x \leftarrow u] \rangle$ then $x \notin \text{lfv}(C'')$.

Lemma 2 (LO Reduction and LO Contexts). Let $t \rightarrow u$ by reducing a redex C . Then C is a \rightarrow_{LO} step iff C is LO.

Structural Equivalence. A peculiar trait of the LSC is that the rewriting rules do not propagate ES. Therefore, evaluation is usually stable by structural equivalences moving ES around. In this paper we use the following equivalence, including garbage collection (\equiv_{gc}), that we prove to be a strong bisimulation.

Definition 7 (Structural equivalence). The structural equivalence \equiv is the symmetric, reflexive, transitive, and contextual closure of the following axioms:

$$\begin{array}{ll} (lx.t)[y \leftarrow u] \equiv_{\lambda} lx.t[y \leftarrow u] & \text{if } x \notin \text{fv}(u) \\ (tu)[x \leftarrow w] \equiv_{@1} t[x \leftarrow w]u & \text{if } x \notin \text{fv}(u) \\ (tu)[x \leftarrow w] \equiv_{@r} tu[x \leftarrow w] & \text{if } x \notin \text{fv}(t) \\ t[x \leftarrow u][y \leftarrow w] \equiv_{\text{com}} t[y \leftarrow w][x \leftarrow u] & \text{if } y \notin \text{fv}(u) \text{ and } x \notin \text{fv}(w) \\ t[x \leftarrow u][y \leftarrow w] \equiv_{[\cdot]} t[x \leftarrow u[y \leftarrow w]] & \text{if } y \notin \text{fv}(t) \\ t[x \leftarrow u] \equiv_{\text{gc}} t & \text{if } x \notin \text{fv}(t) \\ t[x \leftarrow u] \equiv_{\text{dup}} t_{[y]_x}[x \leftarrow u][y \leftarrow u] & \end{array}$$

In \equiv_{dup} , $t_{[y]_x}$ denotes a term obtained from t by renaming some (possibly none) occurrences of x as y , with y a fresh variable.

Proposition 1 (Structural Equivalence \equiv is a Strong Bisimulation). If $t \equiv u \rightarrow_{\text{LO}} w$ then exists r s.t. $t \rightarrow_{\text{LO}} r \equiv w$ and the steps are either both multiplicative or both exponential.

3 Distilleries

An abstract machine \mathbf{M} is meant to implement a strategy \rightarrow via a *distillation*, i.e. a decoding function $\underline{\cdot}$. A machine has a state s , given by a *code* \bar{t} , i.e. a l -term t without ES and not considered up to α -equivalence, and some data-structures like stacks, dumps, environments, and heaps. The data-structures are used to implement the search for the next \rightarrow -redex and some form of substitution, and they decode to evaluation contexts for \rightarrow . Every state s decodes to a term \underline{s} , having the shape $C_s(\bar{t})$, where \bar{t} is the code currently under evaluation and C_s is the evaluation context given by the data-structures.

A machine computes using transitions, whose union is denoted by \rightsquigarrow , of two types. The *principal* one, denoted by \rightsquigarrow_p , corresponds to the firing of a rule defining \rightarrow , up to structural equivalence \equiv . The *commutative* transitions, denoted by \rightsquigarrow_c , only rearrange the data structures, and on the calculus are either invisible or mapped to \equiv . The terminology reflects a proof-theoretic view, as machine transitions can be seen as cut-elimination steps [8,3]. The transformation of evaluation contexts is formalized in the LSC as a structural equivalence \equiv , which is required to commute with evaluation \rightarrow , i.e. to satisfy

$$\begin{array}{ccc} t \xrightarrow{\quad\quad\quad r} & & t \xrightarrow{\quad\quad\quad r} \\ \equiv & \Rightarrow \exists q \text{ s.t. } & \equiv \\ u & & u \dashrightarrow \circ q \end{array}$$

for each of the rules of \rightarrow , preserving the kind of rule. In fact, this means that \equiv is a *strong bisimulation* (i.e. *one step to one step*) with respect to \rightarrow , that is what we proved in Proposition 1 for the equivalence at work in this paper. Strong bisimulations formalize transformations which are transparent with respect to the behavior, even at the level of complexity, because they can be delayed without affecting the length of evaluation:

Lemma 3 (Postponement of \equiv). *If \equiv is a strong bisimulation, $t (\rightarrow \cup \equiv)^* u$ implies $t \rightarrow^* \equiv u$ and the number and kind of steps of \rightarrow in the two reduction sequences is exactly the same.*

We can finally introduce distilleries, i.e. systems where a strategy \rightarrow simulates a machine \mathbf{M} up to structural equivalence \equiv via the decoding $\underline{\cdot}$.

Definition 8. *A distillery $\mathbf{D} = (\mathbf{M}, \rightarrow, \equiv, \underline{\cdot})$ is given by:*

1. *An abstract machine \mathbf{M} , given by*
 - (a) *a deterministic labeled transition system (lts) \rightsquigarrow over states s , with labels in $\{\mathbf{m}, \mathbf{e}, \mathbf{c}\}$; the transitions labelled by \mathbf{m}, \mathbf{e} are called principal, the others commutative;*
 - (b) *a distinguished class of states deemed initial, in bijection with closed l -terms; from these, the reachable states are obtained by applying \rightsquigarrow^* ;*
2. *a deterministic strategy \rightarrow , i.e., a deterministic lts over the terms of the LSC induced by some strategy on its reduction rules, with labels in $\{\mathbf{m}, \mathbf{e}\}$.*

3. a structural equivalence \equiv on terms which is a strong bisimulation with respect to $\rightarrow\circ$;
4. a decoding function $\underline{\cdot}$ from states to terms whose graph, when restricted to reachable states, is a weak simulation up to \equiv (the commutative transitions are considered as τ actions). More explicitly, for all reachable states:
 - projection of principal transitions: $s \rightsquigarrow_p s'$ implies $\underline{s} \rightarrow\circ_p \equiv \underline{s}'$ for all $p \in \{\mathbf{m}, \mathbf{e}\}$;
 - distillation of commutative transitions: $s \rightsquigarrow_c s'$ implies $\underline{s} \equiv \underline{s}'$.

The simulation property is a minimum requirement, but a stronger form of relationship is usually desirable. Additional hypotheses are required in order to obtain the converse simulation and provide complexity bounds.

Terminology: an execution ρ is a sequence of transitions from an initial state. With $|\rho|$, $|\rho|_p$ and $|\rho|_c$ we denote respectively the length, the number of principal and commutative transitions of ρ , whereas $|t|$ denotes the size of a term t .

Definition 9 (Distillation Qualities). A distillery is

- Reflective when on reachable states:
 - Termination: \rightsquigarrow_c terminates;
 - Progress: if s is final then \underline{s} is a $\rightarrow\circ$ -normal form.
- Bilinear when, given an execution ρ from an initial term t :
 - Execution Length: the number of commutative steps $|\rho|_c$ is linear in both $|t|$ and $|\rho|_p$, i.e. $|\rho|_c \leq c \cdot (1 + |\rho|_p) \cdot |t|$ for some non-zero constant c (when $|\rho|_p = 0$, $O(|t|)$ time is still needed to recognize that t is normal).
 - Commutative: each commutative transition is implementable in $O(1)$ time on a RAM;
 - Principal: each principal transition is implementable in $O(|t|)$ time on a RAM.

A reflective distillery is enough to obtain a weak bisimulation between the strategy $\rightarrow\circ$ and the machine M , up to structural equivalence \equiv (again, the weakness is with respect to commutative transitions). With $|\rho|_{\mathbf{m}}$ and $|\rho|_{\mathbf{e}}$ we denote respectively the number of multiplicative and exponential transitions of ρ .

Theorem 1 (Correctness and Completeness). Let D be a reflective distillery and s an initial state.

1. Simulation up to \equiv : for every execution $\rho : s \rightsquigarrow^* s'$ there is a derivation $d : \underline{s} \rightarrow\circ^* \equiv \underline{s}'$ s.t. $|\rho|_{\mathbf{m}} = |d|_{\mathbf{m}}$ and $|\rho|_{\mathbf{e}} = |d|_{\mathbf{e}}$.
2. Reverse Simulation up to \equiv : for every derivation $d : \underline{s} \rightarrow\circ^* t$ there is an execution $\rho : s \rightsquigarrow^* s'$ s.t. $t \equiv \underline{s}'$ and $|\rho|_{\mathbf{m}} = |d|_{\mathbf{m}}$ and $|\rho|_{\mathbf{e}} = |d|_{\mathbf{e}}$.

Bilinearity, instead, is crucial for the low-level theorem.

Theorem 2 (Low-Level Implementation Theorem). Let $\rightarrow\circ$ be a strategy on terms with ES s.t. there exists a bilinear reflective distillery $D = (M, \rightarrow\circ, \equiv, \underline{\cdot})$. Then a derivation $d : t \rightarrow\circ^* u$ is implementable on RAM machines in $O((1 + |d|) \cdot |t|)$ steps, i.e. bilinear in the size $|t|$ of the initial term and the length $|d|$ of the derivation.

Proof. Given $d : t \multimap^n u$ by Theorem 1.2 there is an execution $\rho : s \rightsquigarrow^* s'$ s.t. $u \equiv s'$ and $|\rho|_p = |d|$. The cost of implementing ρ is the sum of the costs of implementing the commutative and the principal transitions. By bilinearity, $|\rho|_c = O((1 + |\rho|_p) \cdot |t|)$ and so all the commutative transitions in ρ require $O((1 + |\rho|_p) \cdot |t|)$ steps, because a single one takes a constant number of steps. Again by bilinearity, each principal one takes $O(|t|)$, and so all the principal transitions together require $O(|\rho|_p \cdot |t|)$ steps. \square

4 Strengthening the MAM

The machine we are about to introduce implements leftmost-outermost reduction and may therefore be seen as a strong version of the Krivine abstract machine (KAM). However, it differs from the KAM in the fundamental point of using global, as opposed to local, environments. It is therefore more appropriate to say that it is a strong version of the machine we introduced in [3], which we called MAM (Milner abstract machine). Let us briefly recall its definition:

$$\begin{array}{c|c|c} \text{Code} & \text{Stack} & \text{Env} \\ \hline \bar{t}\bar{u} & \pi & E \\ lx.\bar{t} & \bar{u} : \pi & E \\ x & \pi & E \end{array} \rightsquigarrow_{c_1}^{\omega} \begin{array}{c|c|c} \text{Code} & \text{Stack} & \text{Env} \\ \hline \bar{t} & \bar{u} : \pi & E \\ \bar{t} & \pi & [x \leftarrow \bar{u}] : E \\ \bar{t}^\alpha & \pi & E \end{array} \quad \text{if } E(x) = \bar{t}$$

Note that the stack and the environment of the MAM contain *codes*, not *closures* as in the KAM. A global environment indeed circumvents the complex mutually recursive notions of *local environment* and *closure*, at the price of the explicit α -renaming \bar{t}^α which is applied *on the fly* in \rightsquigarrow_e . The price however is negligible, at least theoretically, as the asymptotic complexity of the machine is not affected, see [3] (the same can be said of variable names vs de Bruijn indexes/levels).

We know that the MAM performs *weak* head reduction, whose reduction contexts are (informally) of the form $\langle \cdot \rangle \pi$. This justifies the presence of the stack. It is immediate to extend the MAM so that it performs full head reduction, *i.e.*, so that the head redex is reduced even if it is under an abstraction. Since head contexts are of the form $\Lambda \langle \cdot \rangle \pi$ (with Λ a list of abstractions), we simply add a stack of abstractions Λ and augment the machine with the following transition:

$$\begin{array}{c|c|c} \text{Abs} & \text{Code} & \text{Stack} \\ \hline \Lambda & lx.\bar{t} & \epsilon \end{array} \rightsquigarrow_{c_2}^{\omega} \begin{array}{c|c|c} \text{Abs} & \text{Code} & \text{Stack} \\ \hline x : \Lambda & \bar{t} & \epsilon \end{array} \quad | \quad E$$

The other transitions do not touch the Λ stack.

LO reduction is nothing but iterated head reduction. LO reduction contexts, which we formally introduced in Definition 6, when restricted to the pure *l*-calculus (without ES) are of the form $\Lambda.rC\pi$, where: Λ and π are as above; r , if present, is a neutral term; and C is either $\langle \cdot \rangle$ or, inductively, a LO context. Then LO contexts may be represented by stacks of triples of the form (Λ, r, π) , where r is a neutral term. These stacks of triples will be called *dumps*.

The states of the machine for full LO reduction are as above but augmented with a dump and a *phase* φ , indicating whether we are executing head reduction

(\blacktriangledown) or whether we are backtracking to find the starting point of the next iteration (\blacktriangle). To the above transitions (which do not touch the dump and are always in the \blacktriangledown phase), we add the following:

Abs	x	π	E	Dump	\downarrow	Abs	x	π	E	Dump	\downarrow
Λ				D	\rightsquigarrow_{c_3}	Λ				D	\rightsquigarrow
$x : \Lambda$	\bar{t}	ϵ	E	D	\blacktriangle	\rightsquigarrow_{c_5}	Λ	$lx.\bar{t}$	ϵ	D	\blacktriangle
ϵ	\bar{u}	ϵ	E	$(\Lambda, \bar{t}, \pi) : D$	\blacktriangle	\rightsquigarrow_{c_7}	Λ	$\bar{t}\bar{u}$	π	D	\blacktriangle
Λ	\bar{t}	$\bar{u} : \pi$	E	D	\blacktriangle	\rightsquigarrow_{c_6}	ϵ	\bar{u}	ϵ	E	$(\Lambda, \bar{t}, \pi) : D$

where $E(x) = \perp$ means that the variable x is undefined in the environment E .

In the machine we actually use we join the dump and the Λ stack into the *frame* F , to reduce the number of machine components (the analysis will however somewhat reintroduce the distinction). In the sequel, the reader should bear in mind that a state of the Strong MAM introduced below corresponds to a state of the machine just discussed according to the following correspondence:⁴

$$\begin{array}{ll} \text{Discussed Machine:} & \begin{array}{l} \text{Abs} \left| \begin{array}{l} \text{Code} \\ \Lambda_0 \end{array} \right| \begin{array}{l} \text{Stack} \\ \bar{t} \end{array} \left| \begin{array}{l} \text{Env} \\ \pi \end{array} \right| \begin{array}{l} \text{Dump} \\ (\Lambda_1, \bar{t}_1, \pi_1) : \dots : (\Lambda_n, \bar{t}_n, \pi_n) \end{array} \left| \begin{array}{l} \text{Ph} \\ \varphi \end{array} \right. \\ \downarrow \\ \text{Frame} \end{array} \\ \text{Strong MAM:} & A_0 : (\bar{t}_1, \pi_1) : A_1 : \dots : (\bar{t}_n, \pi_n) : A_n \left| \begin{array}{l} \text{Code} \\ \bar{t} \end{array} \right| \begin{array}{l} \text{Stack} \\ \pi \end{array} \left| \begin{array}{l} \text{Env} \\ E \end{array} \right| \begin{array}{l} \text{Ph} \\ \varphi \end{array} \end{array} \end{array}$$

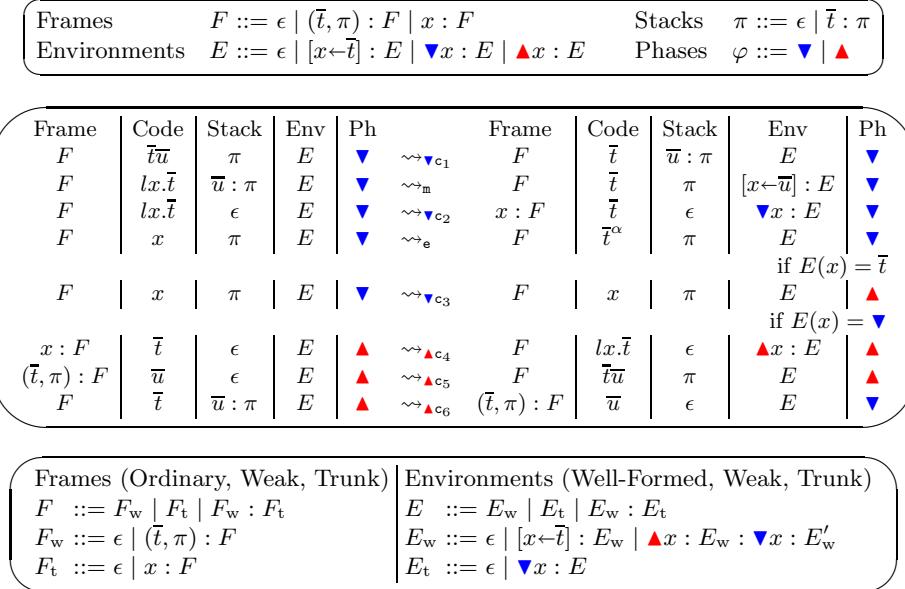
5 The Strong Milner Abstract Machine

The components and the transitions of the Strong MAM are given by the first two boxes in Fig. 1. As above, we use \bar{t}, \bar{u}, \dots to denote *codes*, *i.e.*, terms not containing ES and *well-named*, by which mean that distinct binders bind distinct variables and that the sets of free and bound variables are disjoint (codes are not considered up to α -equivalence). The Strong MAM has two phases: *evaluation* (\blacktriangledown) and *backtracking* (\blacktriangle).

Initial states. The *initial states* of the Strong MAM are of the form $\epsilon \mid \bar{t} \mid \epsilon \mid \epsilon \mid \blacktriangledown$, where \bar{t} is a closed code called the *initial term*. In the sequel, we abusively say that a state is reachable from a term meaning that it is reachable from the corresponding initial state.

Scope Markers. The two transitions to evaluate and backtrack on abstractions, $\rightsquigarrow_{\blacktriangledown c_2}$ and $\rightsquigarrow_{\blacktriangle c_4}$, add markers to delimit subenvironments associated to scopes. The marker $\blacktriangledown x$ is introduced when the machine starts evaluating under an abstraction λx , while $\blacktriangle x$ marks the end of such a subenvironment. Note that the markers are not inspected by the machine. They are in fact needed only for the analysis, as they structure the frame and the environment of a reachable state into *weak* and *trunk* parts, allowing a simple decoding towards terms with ES.

⁴ Modulo the presence of markers of the form $\blacktriangle x$ and $\blacktriangledown x$ in the environment, which are needed for bookkeeping purposes and were omitted here.

**Fig. 1.** The Strong MAM.

Weak and Trunk Frames. A frame F may be uniquely decomposed into $F = F_w : F_t$ (with “ $:$ ” abusively denoting concatenation, as we will always do in the sequel), where $F_w = (\bar{t}_1, \pi_1) : \dots : (\bar{t}_n, \pi_n)$ (with n possibly null) is a *weak frame*, *i.e.* where no abstracted variable appear, and F_t is a *trunk frame*, *i.e.* not of the form $(\bar{t}, \pi) : F'$ (it either starts a variable entry or it is empty). More precisely, we rely on the alternative grammar⁵ in the third box of Fig. 1. We denote by $\Lambda(F)$ the set of variables in F , *i.e.* the set of x s.t. $F = F' : x : F''$.

Weak, Trunk, and Well-Formed Environments. Similarly to the frame, the environment of a reachable state has a weak/trunk structure. In contrast to frames, however, not every environment can be seen this way, but only the well-formed ones (reachable environments will be shown to be well-formed). A weak environment E_w does not contain any open scope, *i.e.* whenever in E_w there is a scope opener marker ($\blacktriangledown x$) then one can also find the scope closer marker ($\blacktriangle x$), and (globally) the closed scopes of E_w are well-parenthesized. A trunk environment E_t may instead also contain open scopes that have no closing marker in E_t (but not unmatched closing markers $\blacktriangle x$). Formally, weak E_w , trunk E_t , and well-formed environments E (all the environments that we will consider will be well-formed, that is why we note them E) are defined in the third box in Fig. 1.

⁵ We slightly abuse notations: the production $F_w : F_t$ may produce $\epsilon : \epsilon$ which is not a valid list/frame. To be formal, one should introduce the composition of lists, noted $F_w \circ F_t$ or $F_t \langle F_w \rangle$ that removes empty frames in excess. To ease the reading, instead, we overload ‘ $:$ ’ with composition.

Accessing Environments and Meta-level Garbage Collection. Fragments of the form $\blacktriangle x : E_w : \blacktriangledown x$ within an environment will essentially be ignored; this is how a simple form of garbage collection is encapsulated at the meta-level in the decoding. In particular, for a well-formed environment E we define $E(x)$ as:

$$\begin{array}{ll} \epsilon(x) := \perp & (\blacktriangle y : E_w : \blacktriangledown y : E)(x) := E(x) \\ ([x \leftarrow \bar{t}] : E)(x) := \bar{t} & (\blacktriangledown x : E)(x) := \blacktriangledown \\ ([y \leftarrow \bar{t}] : E)(x) := E(x) & (\blacktriangledown y : E)(x) := E(x) \end{array}$$

We write $\Lambda(E)$ to denote the set of variables bound to \blacktriangledown by an environment E , *i.e.* those variables whose scope is not closed with \blacktriangle .

Lemma 4 (Weak Environments Contain only Closed Scopes). *If E_w is a weak environment then $\Lambda(E_w) = \emptyset$.*

Implementation. Variables are meant to be implemented as memory locations, so that the environment is simply a store, and accessing it takes constant time on RAM. In particular both the list structure of environments and the scope markers are used to define the decoding (*i.e.* for the analysis), but are not meant to be part of the actual implementation. This is to be kept in mind for the sake of the bilinearity of the distillery to be defined.

Compatibility. In the Strong MAM, both the frame and the environment record information about the abstractions in which evaluation is currently taking place. Clearly, such information has to be coherent, otherwise the decoding of a state becomes impossible. The following compatibility predicate captures the correlation between the structure of the frame and that of the environment.

Definition 10 (Compatibility $F \propto E$). *Compatibility $F \propto E$ between frames and environments is defined by*

1. Base: $\epsilon \propto \epsilon$;
2. Weak Extension: $(F_w : F_t) \propto (E_w : E_t)$ if $F_t \propto E_t$;
3. Abstraction: $(x : F) \propto (\blacktriangledown x : E)$ if $F \propto E$;

Lemma 5 (Properties of Compatibility).

1. Well-Formed Environments: *if F and E are compatible then E is well-formed.*
2. Factorization: *every compatible pair $F \propto E$ can be written as $(F_w : F_t) \propto (E_w : E_t)$ with $F_t = x : F'$ iff $E_t = \blacktriangledown x : E'$;*
3. Open Scopes Match: $\Lambda(F) = \Lambda(E)$.
4. Compatibility and Weak Structures Commute: *for all F_w and E_w , $F \propto E$ iff $(F_w : F) \propto (E_w : E)$.*

Invariants. The properties of the machine that are needed to prove its correctness and completeness are given by the following invariants.

Lemma 6 (Strong MAM invariants). *Let $s = F \mid \bar{u} \mid \pi \mid E \mid \varphi$ be a state reachable from an initial term \bar{t}_0 . Then:*

1. Compatibility: F and E are compatible, i.e. $F \propto E$.
2. Normal Form:
 - (1) Backtracking Code: if $\varphi = \Delta$, then \bar{u} is normal, and if π is non-empty, then \bar{u} is neutral;
 - (2) Frame: if $F = F' : (\bar{w}, \pi') : F''$, then \bar{w} is neutral.
3. Backtracking Free Variables:
 - (1) Backtracking Code: if $\varphi = \Delta$ then $\text{fv}(\bar{u}) \subseteq \Lambda(F)$;
 - (2) Pairs in the Frame: if $F = F' : (\bar{w}, \pi') : F''$ then $\text{fv}(\bar{w}) \subseteq \Lambda(F'')$.
4. Name:
 - (1) Substitutions: if $E = E' : [x \leftarrow \bar{t}] : E''$ then x is fresh wrt \bar{t} and E'' ;
 - (2) Markers: if $E = E' : \nabla x : E''$ and $F = F' : x : F''$ then x is fresh wrt E'' and F'' , and $E'(y) = \perp$ for any free variable y in F'' ;
 - (3) Abstractions: if $\lambda x. \bar{t}$ is a subterm of F , \bar{u} , π , or E then x may occur only in \bar{t} and in the closed subenvironment $\Delta x : E_w : \nabla x$ of E , if it exists.
5. Closure:
 - (1) Environment: if $E = E' : [x \leftarrow \bar{t}] : E''$ then $E''(y) \neq \perp$ for all $y \in \text{fv}(\bar{t})$;
 - (2) Code, Stack, and Frame: $E(x) \neq \perp$ for any free variable in \bar{u} and in any code of π and F .

Since the statement of the invariants is rather technical, let us summarize the dependencies (or lack thereof) of the various points and their use in the distillation proof of the next section.

- The compatibility, normal form and backtracking free variables invariants are independent of each other and of the subsequent invariants.
- The name invariant relies on the compatibility invariant only. It implies the determinism of the machine (because in the variable case at most one among \rightsquigarrow_e and $\rightsquigarrow_{\Delta c_4}$ applies).
- The closure invariant relies on the compatibility, name and backtracking free variable invariants only. It is crucial for the progress property (because in the variable case at least one among \rightsquigarrow_e and $\rightsquigarrow_{\Delta c_4}$ applies).

The proof of every invariant is by induction on the number of transitions leading to the reachable state. In this respect, the various points of the statement of each invariant are entangled, in the sense that each point needs to use the induction hypothesis of one of the other points, and thus they cannot be proved separately.

6 Distilling the Strong MAM

The definition of the decoding relies on the notion of compatible pair.

Definition 11 (Decoding). Let $s = (F, \bar{t}, \pi, E, \varphi)$ be a state s.t. $F \propto E$ is a compatible pair. Then s decodes to a state context C_s and a term \underline{s} as follows:

Weak Environments:	Compatible Pairs:	
$\underline{\epsilon} := \langle \cdot \rangle$	$\underline{\epsilon \propto \epsilon} := \langle \cdot \rangle$	
$\underline{[x \leftarrow \bar{u}] : E_w} := \underline{E_w} \langle \langle \cdot \rangle [x \leftarrow \bar{u}] \rangle$	$\underline{(F_w : F_t) \propto (E_w : E_t)} := \underline{F_t \propto E_t} \langle \underline{E_w} \langle F_w \rangle \rangle$	
$\underline{\Delta x : E_w : \nabla x : E'_w} := \underline{E'_w}$	$\underline{(x : F) \propto (\nabla x : E)} := \underline{F \propto E} \langle \underline{l x} \langle \cdot \rangle \rangle$	
Weak Frames:	Stacks:	States:
$\underline{\epsilon} := \langle \cdot \rangle$	$\underline{\epsilon} := \langle \cdot \rangle$	$C_s := \underline{F \propto E} \langle \underline{\pi} \rangle$
$\underline{(\bar{u}, \pi) : F_w} := \underline{F_w} \langle \underline{\pi} \langle \bar{u} \langle \cdot \rangle \rangle \rangle$	$\underline{\bar{u} : \pi} := \underline{\pi} \langle \langle \cdot \rangle \bar{u} \rangle$	$\underline{s} := C_s \langle \bar{t} \rangle$

The following lemmas sum up the properties of the decoding.

Lemma 7 (Closed Scopes Disappear). Let $F \propto E$ be a compatible pair. Then $\underline{F \propto (\Delta x : E_w : \nabla x : E)} = \underline{F \propto E}$.

Lemma 8 (LO Decoding Invariant). Let $s = F \mid \bar{u} \mid \pi \mid E \mid \varphi$ be a reachable state. Then $\underline{F \propto E}$ and C_s are LO contexts.

Lemma 9 (Decoding and Structural Equivalence \equiv).

1. Stacks and Substitutions Commute: if x does not occur free in π then $\underline{\pi} \langle t[x \leftarrow u] \rangle \equiv \underline{\pi} \langle t \rangle [x \leftarrow u]$;
2. Compatible Pairs Absorb Substitutions: if x does not occur free in F then $\underline{F \propto E} \langle t[x \leftarrow u] \rangle \equiv \underline{F \propto ([x \leftarrow u] : E)} \langle t \rangle$.

The next theorem is our first main result. By the abstract approach presented in Sect. 3 (Theorem 1), it implies that the Strong MAM is a correct and complete implementation of linear LO evaluation to normal form.

Theorem 3 (Distillation). (*Strong MAM, \rightarrow_{LO} , \equiv , $\underline{\cdot}$*) is an explicit and reflective distillery. In particular:

1. Projection of Principal Transitions:
 - (a) Multiplicative: if $s \rightsquigarrow_m s'$ then $\underline{s} \rightarrow_m \equiv \underline{s}'$;
 - (b) Exponential: if $s \rightsquigarrow_e s'$ then $\underline{s} \rightarrow_e \underline{s}'$, duplicating the same subterm.
2. Distillation of Commutative Transitions:
 - (a) Garbage Collection of Weak Environments: if $s \rightsquigarrow_{c_4} s'$ then $\underline{s} \equiv_{gc} \underline{s}'$;
 - (b) Equality Cases: if $s \rightsquigarrow_{c_{1,2,3,5,6}} s'$ then $\underline{s} = \underline{s}'$.

Proof. Recall, the decoding is defined as $\underline{(F, \bar{t}, \pi, E, \varphi)} := \underline{F \propto E} \langle \underline{\pi} \langle \bar{t} \rangle \rangle$. Determinism of the machine follows by the name invariant (Lemma 6.4), and that of the strategy follows from the totality of the LO order (Lemma 1). We list all cases but the simple equality ones, which may be found in Appendix D.4.

- **Case** $s = (F, lx.\bar{t}, \bar{u} : \pi, E, \blacktriangledown) \rightsquigarrow_{\text{m}} (F, \bar{t}, \pi, [x \leftarrow \bar{u}] : E, \blacktriangledown) = s'$. Note that $C_{s'} = \underline{F \propto E}(\underline{\pi})$ is LO by the LO decoding invariant (Lemma 8). Moreover by the closure invariant (Lemma 6.5) x does not occur in F nor π , justifying the use of Lemma 9 in:

$$\begin{aligned}
 \underline{(F, lx.\bar{t}, \bar{u} : \pi, E, \blacktriangledown)} &= \underline{\frac{F \propto E \langle \bar{u} : \pi \langle lx.\bar{t} \rangle}{F \propto E \langle \underline{\pi} \langle lx.\bar{t} \rangle \bar{u} \rangle}} \\
 &= \underline{\frac{F \propto E \langle \underline{\pi} \langle \bar{t} \rangle [x \leftarrow \bar{u}] \rangle}{F \propto E \langle \underline{\pi} \langle \bar{t} \rangle \rangle}} \\
 &\stackrel{\text{L.9.1}}{=} \underline{F \propto E \langle \underline{\pi} \langle \bar{t} \rangle [x \leftarrow \bar{u}] \rangle} \\
 &\stackrel{\text{L.9.2}}{=} \underline{F \propto ([x \leftarrow \bar{u}] : E) \langle \underline{\pi} \langle \bar{t} \rangle \rangle} = \underline{(F, \bar{t}, \pi, [x \leftarrow \bar{u}] : E, \blacktriangledown)}
 \end{aligned}$$

- **Case** $s = (F, x, \pi, E, \blacktriangledown) \rightsquigarrow_{\text{e}} (F, \bar{t}^\alpha, \pi, E, \blacktriangledown) = s'$ with $E(x) = \bar{t}$. As before, C_s is LO by Lemma 8. Moreover, $E(x) = \bar{t}$ guarantees that E , and thus C_s , have a substitution binding x to \bar{t} . Finally, $C_s = C_{s'}$. Then

$$\underline{s} = C_s \langle x \rangle \rightarrow_{\text{e}} C_s \langle \bar{t}^\alpha \rangle = \underline{s}'$$

- **Case** $s = (x : F, \bar{t}, \epsilon, E, \blacktriangle) \rightsquigarrow_{\blacktriangle c_4} (F, lx.\bar{t}, \epsilon, \blacktriangle x : E, \blacktriangle) = s'$. By Lemma 6.1 $x : F \propto E$, and by Lemma 5.2 $E = E_w : \blacktriangledown x : E'$. Then

$$(x : F) \propto E = (x : F) \propto (E_w : \blacktriangledown x : E') = (x : F) \propto (\blacktriangledown x : E') \langle E_w \rangle$$

Since we are in a backtracking phase (\blacktriangle), the backtracking free variables invariant (Lemma 6.3.1) and the open scopes matching property (Lemma 5.3) give $\text{fv}(\bar{t}) \subseteq_{\text{L.6.1}} \Lambda(F) =_{\text{L.5.3}} \Lambda(E_w : \blacktriangledown x : E') =_{\text{L.4}} \Lambda(\blacktriangledown x : E')$, i.e. E_w does not bind any variable in $\text{fv}(\bar{t})$. Then $\underline{E_w} \langle \bar{t} \rangle \equiv_{\text{gc}}^* \bar{t}$, and

$$\begin{aligned}
 \underline{(x : F, \bar{t}, \epsilon, E, \blacktriangle)} &= \underline{(x : F) \propto E \langle \bar{t} \rangle} \\
 &= \underline{(x : F) \propto (E_w : \blacktriangledown x : E') \langle \bar{t} \rangle} \\
 &= \underline{(x : F) \propto (\blacktriangledown x : E') \langle E_w \langle \bar{t} \rangle \rangle} \\
 &\stackrel{\text{gc}}{=} \underline{(x : F) \propto (\blacktriangledown x : E') \langle \bar{t} \rangle} \\
 &= \underline{F \propto E' \langle \lambda x. \bar{t} \rangle} \\
 &\stackrel{\text{L.7}}{=} \underline{F \propto (\blacktriangle x : E_w : \blacktriangledown x : E') \langle lx. \bar{t} \rangle} \\
 &= \underline{F \propto (\blacktriangle x : E) \langle lx. \bar{t} \rangle} = \underline{(F, lx. \bar{t}, \epsilon, \blacktriangle x : E, \blacktriangle)}
 \end{aligned}$$

For what concerns reflectiveness, *termination* of commutative transitions is subsumed by bilinearity (Theorem 4 below). For *progress*, note that

1. *the machine cannot get stuck during the evaluation phase*: for applications and abstractions it is evident and for variables one among $\rightsquigarrow_{\text{e}}$ and $\rightsquigarrow_{\blacktriangle c_3}$ always applies, because of the closure invariant (Lemma 6.5).
2. *final states have the form* $(\epsilon, t, \epsilon, E, \blacktriangle)$, because
 - (a) by the previous consideration they are in a backtracking phase,
 - (b) if the stack is non-empty then $\rightsquigarrow_{\blacktriangle c_6}$ applies,
 - (c) otherwise if the frame is not empty then either $\rightsquigarrow_{\blacktriangle c_4}$ or $\rightsquigarrow_{\blacktriangle c_5}$ applies.
3. *final states decode to normal terms*: a final state $s = (\epsilon, t, \epsilon, E, \blacktriangle)$ decodes to $\underline{s} = \underline{E} \langle t \rangle$ which is normal and closed by the normal form (Lemma 6.2.1) and backtracking free variables (Lemma 6.3.1) invariants. \square

7 Complexity Analysis

The complexity analysis requires a further invariant, bounding the size of the duplicated subterms. For us, \bar{u} is a subterm of \bar{t} if it does so up to variable names, both free and bound. More precisely: define t^- as t in which all variables (including those appearing in binders) are replaced by a fixed symbol $*$. Then, we will consider u to be a subterm of t whenever u^- is a subterm of t^- in the usual sense. The key property ensured by this definition is that the size $|\bar{u}|$ of \bar{u} is bounded by $|\bar{t}|$.

Lemma 10 (Subterm Invariant). *Let ρ be an execution from an initial code \bar{t} . Every code duplicated along ρ using \rightsquigarrow_e is a subterm of \bar{t} .*

Via the distillation theorem (Theorem 3), the invariant provides a new proof of the subterm property of linear LO reduction (first proved in [5]).

Lemma 11 (Subterm Property for \rightarrow_{LO}). *Let d be a \rightarrow_{LO} -derivation from an initial term t . Every term duplicated along d using \rightarrow_e is a subterm of t .*

The next theorem is our second main result, from which the low-level implementation theorem (Theorem 2) follows. Let us stress that, despite the simplicity of the reasoning, the analysis is subtle as the length of backtracking phases (Point 2) can be bound only *globally*, by the whole previous evaluation work.

Theorem 4 (Bilinearity). *The Strong MAM is bilinear, i.e. given an execution $\rho : s \rightsquigarrow^* s'$ from an initial state of code t then:*

1. Commutative Evaluation Steps are Bilinear: $|\rho|_{\nabla_c} \leq (1 + |\rho|_e) \cdot |t|$.
2. Commutative Evaluation Bounds Backtracking: $|\rho|_{\Delta_c} \leq 2 \cdot |\rho|_{\nabla_c}$.
3. Commutative Steps are Bilinear: $|\rho|_c \leq 3 \cdot (1 + |\rho|_e) \cdot |t|$.

Proof. 1. We prove a slightly stronger statement, namely $|\rho|_{\nabla_c} + |\rho|_m \leq (1 + |\rho|_e) \cdot |t|$, by means of the following notion of size for stacks/frames/states:

$$\begin{array}{ll} |\epsilon| := 0 & |x : F| := |F| \\ |\bar{t} : \pi| := |\bar{t}| + |\pi| & |(\bar{t}, \pi) : F| := |\pi| + |F| \\ |(F, \bar{t}, \pi, E, \nabla)| := |F| + |\pi| + |\bar{t}| & |(F, \bar{t}, \pi, E, \Delta)| := |F| + |\pi| \end{array}$$

By direct inspection of the rules of the machine it can be checked that:

- *Exponentials Increase the Size*: if $s \rightsquigarrow_e s'$ is an exponential transition, then $|s'| \leq |s| + |t|$ where $|t|$ is the size of the initial term; this is a consequence of the fact that exponential steps retrieve a piece of code from the environment, which is a subterm of the initial term by Lemma 10;
- *Non-Exponential Evaluation Transitions Decrease the Size*: if $s \rightsquigarrow_a s'$ with $a \in \{\mathbf{m}, \nabla c_1, \nabla c_2, \nabla c_3\}$ then $|s'| < |s|$;
- *Backtracking Transitions do not Change the Size*: if $s \rightsquigarrow_a s'$ with $a \in \{\Delta c_4, \Delta c_5, \Delta c_6\}$ then $|s'| = |s|$.

Then a straightforward induction on $|\rho|$ shows that

$$|s'| \leq |s| + |\rho|_e \cdot |t| - |\rho|_{\text{v}_c} - |\rho|_m$$

i.e. that $|\rho|_{\text{v}_c} + |\rho|_m \leq |s| + |\rho|_e \cdot |t| - |s'|$.

Now note that $|\cdot|$ is always non-negative and that since s is initial we have $|s| = |t|$. We can then conclude with

$$\begin{aligned} |\rho|_{\text{v}_c} + |\rho|_m &\leq |s| + |\rho|_e \cdot |t| - |s'| \\ &\leq |s| + |\rho|_e \cdot |t| = |t| + |\rho|_e \cdot |t| = (1 + |\rho|_e) \cdot |t| \end{aligned}$$

2. We have to estimate $|\rho|_{\Delta_c} = |\rho|_{\Delta c_4} + |\rho|_{\Delta c_5} + |\rho|_{\Delta c_6}$. Note that
 - (a) $|\rho|_{\Delta c_4} \leq |\rho|_{\text{v}_{c_2}}$, as $\rightsquigarrow_{\Delta c_4}$ pops variables from F , pushed only by $\rightsquigarrow_{\text{v}_{c_2}}$;
 - (b) $|\rho|_{\Delta c_5} \leq |\rho|_{\Delta c_6}$, as $\rightsquigarrow_{\Delta c_5}$ pops pairs (\bar{t}, π) from F , pushed only by $\rightsquigarrow_{\Delta c_6}$;
 - (c) $|\rho|_{\Delta c_6} \leq |\rho|_{\text{v}_{c_3}}$, as $\rightsquigarrow_{\Delta c_6}$ ends backtracking phases, started only by $\rightsquigarrow_{\text{v}_{c_3}}$.
 Then $|\rho|_{\Delta_c} \leq |\rho|_{\text{v}_{c_2}} + 2|\rho|_{\text{v}_{c_3}} \leq 2|\rho|_{\text{v}_c}$.
3. We have $|\rho|_c = |\rho|_{\text{v}_c} + |\rho|_{\Delta_c} \leq_{P.2} |\rho|_{\text{v}_c} + 2|\rho|_{\text{v}_c} =_{P.1} 3 \cdot (1 + |\rho|_e) \cdot |t|$.
 Last, every transition but \rightsquigarrow_e takes a constant time on a RAM. The renaming in a \rightsquigarrow_e step is instead linear in $|\bar{t}|$, by the subterm invariant (Lemma 10). \square

Acknowledgments. This work was partially supported by projects LOGI ANR-2010-BLAN-0213-02, COQUAS ANR-12-JS02-006-01, ELICA ANR-14-CE25-0005, the Saint-Exupéry program funded by the French embassy and the Ministry of Education in Argentina, the French–Argentinian laboratory in Computer Science INFINIS, and the French Argentinian project ECOS-Sud A12E04.

References

1. Abramsky, S., Ong, C.L.: Full abstraction in the lazy lambda calculus. *Inf. Comput.* 105(2), 159–267 (1993)
2. Accattoli, B.: An abstract factorization theorem for explicit substitutions. In: RTA. pp. 6–21 (2012)
3. Accattoli, B., Barenbaum, P., Mazza, D.: Distilling abstract machines. In: ICFP 2014. pp. 363–376 (2014)
4. Accattoli, B., Bonelli, E., Kesner, D., Lombardi, C.: A nonstandard standardization theorem. In: POPL. pp. 659–670 (2014)
5. Accattoli, B., Dal Lago, U.: Beta Reduction is Invariant, Indeed. In: CSL-LICS 2014. p. 8 (2014)
6. Accattoli, B., Sacerdoti Coen, C.: On the value of variables. In: WoLLIC 2014. pp. 36–50 (2014)
7. Accattoli, B., Sacerdoti Coen, C.: On the relative usefulness of fireballs. Accepted at LICS 2015 (2015)
8. Ariola, Z.M., Bohannon, A., Sabry, A.: Sequent calculi and abstract machines. *ACM Trans. Program. Lang. Syst.* 31(4) (2009)
9. Biernacka, M., Danvy, O.: A concrete framework for environment machines. *ACM Trans. Comput. Log.* 9(1) (2007)
10. Boutiller, P.: De nouveaux outils pour manipuler les inductif en Coq. Ph.D. thesis, Université Paris Diderot - Paris 7 (2014)

11. de Carvalho, D.: Execution time of lambda-terms via denotational semantics and intersection types. CoRR abs/0905.4251 (2009)
12. Crégut, P.: An abstract machine for lambda-terms normalization. In: LISP and Functional Programming. pp. 333–340 (1990)
13. Crégut, P.: Strongly reducing variants of the Krivine abstract machine. Higher-Order and Symbolic Computation 20(3), 209–230 (2007)
14. Curien, P.: An abstract framework for environment machines. Theor. Comput. Sci. 82(2), 389–402 (1991)
15. Danos, V., Regnier, L.: Head linear reduction. Tech. rep. (2004)
16. Danvy, O., Nielsen, L.R.: Refocusing in reduction semantics. Tech. Rep. RS-04-26, BRICS (2004)
17. Danvy, O., Zerny, I.: A synthetic operational account of call-by-need evaluation. In: PPDP. pp. 97–108 (2013)
18. Dénès, M.: Étude formelle d’algorithmes efficaces en algèbre linéaire. Ph.D. thesis, Université de Nice - Sophia Antipolis (2013)
19. Ehrhard, T., Regnier, L.: Böhm trees, Krivine’s machine and the Taylor expansion of lambda-terms. In: CiE. pp. 186–197 (2006)
20. Fernández, M., Siafakas, N.: New developments in environment machines. Electr. Notes Theor. Comput. Sci. 237, 57–73 (2009)
21. García-Pérez, Á., Nogueira, P., Moreno-Navarro, J.J.: Deriving the full-reducing krivine machine from the small-step operational semantics of normal order. In: PPDP. pp. 85–96 (2013)
22. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: (ICFP ‘02). pp. 235–246 (2002)
23. Hardin, T., Maranget, L.: Functional runtime systems within the lambda-sigma calculus. J. Funct. Program. 8(2), 131–176 (1998)
24. Lang, F.: Explaining the lazy Krivine machine using explicit substitution and addresses. Higher-Order and Symbolic Computation 20(3), 257–270 (2007)
25. Mascari, G., Pedicini, M.: Head linear reduction and pure proof net extraction. Theor. Comput. Sci. 135(1), 111–137 (1994)
26. Milner, R.: Local bigraphs and confluence: Two conjectures. Electr. Notes Theor. Comput. Sci. 175(3), 65–73 (2007)
27. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. 1(2), 125–159 (1975)
28. Sands, D., Gustavsson, J., Moran, A.: Lambda calculi and linear speedups. In: The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones. pp. 60–84 (2002)
29. Smith, C.: Abstract machines for higher-order term sharing, Presented at IFL 2014

A Proofs Omitted from Sect. 2 (Linear Leftmost-Outermost Reduction)

The proofs omitted from Sect. 2 are:

1. Lemma 1, stating the totality of the \prec_{LO} order. The proof is a trivial induction on t .
2. Lemma 2, stating the equivalence of LO contexts and LO reduction. It is proved in the next subsection.
3. Proposition 1, stating that structural equivalence \equiv is a strong bisimulation. The very long and tedious proof is postponed to the last section of the appendix, at page 31.

A.1 Proof of the Equivalence of Definitions for LO Contexts (Lemma 2)

Proof.

$\Rightarrow)$ There are three cases:

(a) *Left application*: if $C = C' \langle C'' t \rangle$ then clearly $C'' \neq L \langle \lambda x. C''' \rangle$, otherwise C is not the position of the LO redex.

(b) *Right Application*: let $C = C' \langle w C'' \rangle$, and note w is neutral otherwise C is not the position of the LO redex.

(c) *Substitution*: if $C = C' \langle C''[x \leftarrow u] \rangle$ then $x \notin \text{lfv}(C'')$ otherwise there is an exponential redex of position $\prec_{LO} C$, which would be absurd.

$\Leftarrow)$ Let C' the position of the \rightarrow_{LO} step in t and suppose, for the sake of absurdity, that $C' \neq C$. By definition $C' \prec_{LO} C$. We have two cases:

(a) $C' \prec_O C$. Then necessarily C' identifies a \rightarrow_m -redex and we have $C = C' \langle L \langle \lambda x. C'' \rangle w \rangle$. It follows that C is not a LO context, because the left application clause is contradicted, absurd.

(b) $C' \prec_L C$. Then there is a decomposition $C = C'' \langle w C''' \rangle$ with the hole of C' falling in w . By hypothesis w is neutral. Then $w = C_0 \langle x \rangle$ and the \rightarrow_{LO} step is a \rightarrow_e -step substituting on x from a substitution in C'' , i.e. $C'' = C^\bullet \langle C^\circ[x \leftarrow t] \rangle$ for some contexts C^\bullet and C° . Then $C = C^\bullet \langle C^\circ \langle w C''' \rangle [x \leftarrow t] \rangle$ and $x \in \text{lfv}(C^\circ \langle w C''' \rangle)$, which contradicts the substitution clause in the hypothesis that C is a LO context. \square

B Proofs Omitted from Sect. 3 (Distilleries)

The proof of Lemma 3, stating that a strong bisimulation \equiv can be postponed, is a straightforward induction on the number of rewriting steps in t ($\multimap \cup \equiv$) * u .

The proof of Theorem 1, stating the correctness and completeness of the implementation for a reflective distillery, follows. The simulation is a simple proof by induction using the postponement lemma, while the reverse simulation is a similar induction following from the properties of a reflective distillery and by determinism of \multimap .

Proof (of Theorem 1).

1. *Strong Simulation:* by induction on the length of ρ . If ρ is empty then the empty derivation satisfies the statement. If ρ is given by $\sigma : s \rightsquigarrow^{k-1} s''$ followed by $s'' \rightsquigarrow s'$. By *i.h.* there exists $e : \underline{s} \multimap^* \underline{s}''$ s.t. $|\sigma|_p = |e|$. Cases of $s'' \rightsquigarrow s'$:
 - (a) *Principal:* by definition of a distillery, $\underline{s}'' \multimap \underline{s}'$, and so $\underline{s} \multimap^* \underline{s}'' \multimap \underline{s}'$. By the postponement lemma (Lemma 3) the use of \equiv between \multimap^* and \multimap can be postponed, obtaining a term u and a derivation d s. t. $d : \underline{s} \multimap^* u \multimap \underline{s}'$ with $|d| = |e| + 1 =_{i.h.} |\sigma|_p + 1 = |\rho|_p$.
 - (b) *Commutative:* by definition of a distillery, $\underline{s}'' \equiv \underline{s}'$, and so $d : \underline{s} \multimap^* \underline{s}'' \equiv \underline{s}'$ verifies $|d| = |e| =_{i.h.} |\sigma|_p = |\rho|_p$.
2. *Reverse Strong Simulation:* we use $\text{nf}_c(s)$ to denote the commutative normal form of s , that exists and is unique because by hypothesis \rightsquigarrow_c terminates and the machine is deterministic. The proof is by induction on the length of d . If d is empty then the empty execution satisfies the statement.
 If d is given by $e : \underline{s} \multimap^* u$ followed by $u \multimap t$ then by *i.h.* there is an execution $\sigma : s \rightsquigarrow^* s''$ s.t. $u \equiv \underline{s}''$ and $|\sigma|_p = |e|$. Note that since commutative transitions are distilled away, σ can be extended as $\sigma' : s \rightsquigarrow^* s'' \rightsquigarrow_c^* \text{nf}_c(s'')$ with $u \equiv \underline{\text{nf}_c(s'')}$ and $|\sigma'|_p = |e|$. Now, if $u \multimap t$ then $\text{nf}_c(s'')$ cannot be a final state, otherwise there would be a contradiction with the progress hypothesis for a reflective distillery. Then $\text{nf}_c(s'') \rightsquigarrow_p s'$ (the transition cannot be commutative because $\text{nf}_c(s'')$ is a commutative normal form). Now, by definition of distillery there exists w s.t. $\underline{\text{nf}_c(s'')} \multimap w \equiv \underline{s}'$. But $u \equiv \underline{\text{nf}_c(s'')} \multimap w$, so by Lemma 3 there exists t' s.t. $u \multimap t' \equiv w \equiv \underline{s}'$. Now the determinism of \multimap implies $t' = t$, allowing us to conclude. \square

C Proofs Omitted from Sect. 5 (The Strong Milner Abstract Machine)

First of all, Lemma 4 (namely: *If E_w is a weak environment then $\Lambda(E_w) = \emptyset$*) is proved by a straightforward induction on the definition of weak environment E_w .

Then we prove the properties of compatibility (next subsection), and the invariants (Lemma 6). The proof of every invariant is studied separately, to stress the dependencies wrt to other invariants.

C.1 Proof of the Properties of Compatibility (Lemma 5)

Proof. The first three points (well-formed environments, factorization, open scopes) are by induction on the definition of compatible pair, and well-formed environments is omitted because it is evident. The fourth case is rather a corollary of factorization, and will be treated after the induction. The base case of the inductive reasoning is immediate for both factorization and open scopes. Two inductive cases:

1. *Weak Extension:*

- (a) *Factorization:* the decomposition is immediate, and the correspondence about the first variable name follows from the *i.h.*.
- (b) *Open Scopes:* by *i.h.*, $\Lambda(F_t) = \Lambda(E_t)$. By Lemma 4, $\Lambda(E_w) = \emptyset$, and by definition $\Lambda(F_w) = \emptyset$. Then $\Lambda(F) = \Lambda(F_w) \cup \Lambda(F_t) = \Lambda(F_t) = \Lambda(E_t) = \Lambda(E_w) \cup \Lambda(E_t) = \Lambda(E)$.

2. *Abstraction*

- (a) *Factorization:* by definition $x : F$ and $\nabla x : E$ are a trunk frame F_t and a trunk environment E_t , respectively. given that $:$ is overloaded with composition, and weak trunk and environments can be empty we have $F_t =: F$, and similarly for E_t , proving the decomposition property. The correspondence about the first variable name is evident.
- (b) *Open Scopes:* $\Lambda(x : F) = \{x\} \cup \Lambda(F) =_{i.h.} \{x\} \cup \Lambda(E) = \Lambda(x : E)$.

Compatibility and Weak Structures Commute:

1. $\Rightarrow)$ By factorization (Point 2), $F = F'_w : F_t$ and $E = E'_w : E_t$. By definition of compatibility, if $F \propto E$ is derivable then $F_t \propto E_t$ is also derivable. Now $F_w : F'_w$ and $E_w : E'_w$ are weak structures and so by the weak extension rule $F_w : F = F_w : F'_w : F_t \propto E_w : E'_w : E_t = E_w : E$.
2. $\Leftarrow)$ By definition of compatibility, if $F_w : F = F_w : F'_w : F_t \propto E_w : E'_w : E_t = E_w : E$ is derivable then $F_t \propto E_t$ is also derivable, and $F = F'_w : F_t \propto E'_w : E_t = E$ by applying the weak extension rule. \square

C.2 Proof of the Compatibility Invariant (Lemma 6.1)

Proof. By induction on the length of the number of transitions to reach s . The invariant trivially holds for an initial state. For a non-empty evaluation sequence we list the cases for the last transitions. We only deal with those that act on the frame or on the environment, as the others immediately follows from the *i.h.*.

- **Case** $(F, l x. \bar{t}, \bar{u} : \pi, E, \nabla) \rightsquigarrow_m (F, \bar{t}, \pi, [x \leftarrow \bar{u}] : E, \nabla)$. By *i.h.* F and E are compatible, *i.e.* $F = (F_w : F_t) \propto (E_w : E_t) = E$ with $F_t \propto E_t$. Since $[x \leftarrow \bar{u}] : E_w$ is still a weak environment, we have $(F_w : F_t) \propto ([x \leftarrow \bar{u}] : E_w : E_t)$, *i.e.* $F \propto ([x \leftarrow \bar{u}] : E)$.
- **Case** $(F, l x. \bar{t}, \epsilon, E, \nabla) \rightsquigarrow_{\nabla c_2} (x : F, \bar{t}, \epsilon, \nabla x : E, \nabla)$. By *i.h.* $F \propto E$. By definition of compatibility we obtain $(x : F) \propto (\nabla x : E)$.
- **Case** $(x : F, \bar{t}, \epsilon, E, \Delta) \rightsquigarrow_{\Delta c_4} (F, l x. \bar{t}, \epsilon, \Delta x : E, \Delta)$. By *i.h.*, $(x : F) \propto E$. By the factorization property of compatible pairs (Lemma 5.2) $E = E'_w : \nabla x : E'$ with $F \propto E'$. Now $\Delta x : E = \Delta x : E_w : \nabla x : E' = E'_w : E'$. Then, from $F \propto E'$ by definition $F \propto (E'_w : E')$, *i.e.* $F \propto (\Delta x : E)$.
- **Case** $((\bar{t}, \pi) : F, \bar{u}, \epsilon, E, \Delta) \rightsquigarrow_{\Delta c_5} (F, \bar{t}\bar{u}, \pi, E, \Delta)$. By *i.h.*, $((\bar{t}, \pi) : F) \propto E$, so $F \propto E$ by Lemma 5.4.
- **Case** $(F, \bar{t}, \bar{u} : \pi, E, \Delta) \rightsquigarrow_{\Delta c_6} ((\bar{t}, \pi) : F, \bar{u}, \epsilon, E, \nabla)$. By *i.h.*, we have that $F \propto E$ which implies $((\bar{t}, \pi) : F) \propto E$ by Lemma 5.4. \square

C.3 Proof of the Normal Form Invariant (Lemma 6.2)

Proof. The invariant trivially holds for an initial state $\epsilon \mid \bar{t} \mid \epsilon \mid \epsilon \mid \blacktriangledown$. For a non-empty evaluation sequence we list the cases for the last transitions. We only consider the cases for backtracking phases (\blacktriangle) or when the frame changes, the others ($\rightsquigarrow_{\blacktriangledown c_1}, \rightsquigarrow_m, \rightsquigarrow_e$) are omitted because they follow immediately from the *i.h.*.

- **Case** $(F, lx.\bar{t}, \epsilon, E, \blacktriangledown) \rightsquigarrow_{\blacktriangledown c_2} (x : F, \bar{t}, \epsilon, \blacktriangledown x : E, \blacktriangledown)$.
 1. Trivial since $\varphi \neq \blacktriangle$.
 2. Suppose $x : F$ can be written as $x : F' : (\bar{u}, \pi') : F''$. Then by *i.h.* \bar{u} is a neutral term.
- **Case** $(F, x, \pi, E, \blacktriangledown) \rightsquigarrow_{\blacktriangledown c_3} (F, x, \pi, E, \blacktriangle)$ **with** $E(x) = \blacktriangledown$. Note that $x \in \Lambda(E)$, because $E(x) = \blacktriangledown$.
 1. x is a normal and neutral term.
 2. It follows from the *i.h.*, as F is unchanged.
- **Case** $(x : F, \bar{t}, \epsilon, E, \blacktriangle) \rightsquigarrow_{\blacktriangle c_4} (F, lx.\bar{t}, \epsilon, \blacktriangle x : E, \blacktriangle)$.
 1. By *i.h.* we know that \bar{t} is a normal form. Then $lx.\bar{t}$ is a normal form. the stack is empty, so we conclude.
 2. It follows from the *i.h.*.
- **Case** $((\bar{t}, \pi) : F, \bar{u}, \epsilon, E, \blacktriangle) \rightsquigarrow_{\blacktriangle c_5} (F, \bar{t}\bar{u}, \pi, E, \blacktriangle)$.
 1. By *i.h.* we have that \bar{u} is a normal term while by Point 2 of the *i.h.* \bar{t} is neutral. Therefore $\bar{t}\bar{u}$ is a neutral term.
 2. It follows from the *i.h..*
- **Case** $(F, \bar{t}, \bar{u} : \pi, E, \blacktriangle) \rightsquigarrow_{\blacktriangle c_6} ((\bar{t}, \pi) : F, \bar{u}, \epsilon, E, \blacktriangledown)$.
 1. Trivial since $\varphi \neq \blacktriangle$.
 2. \bar{t} is a neutral term by Point 1 of the *i.h..* □

C.4 Proof of the Backtracking Free Variables Invariant (Lemma 6.3)

Proof. The invariant trivially holds for an initial state $\epsilon \mid \bar{t}_0 \mid \epsilon \mid \epsilon \mid \blacktriangledown$ if \bar{t}_0 is closed and well-named. For a non-empty evaluation sequence we list the cases for the last transitions. We omit the transitions involving only states in evaluating phase, as for them everything follows immediately from the *i.h..*

- **Case** $(F, y, \pi, E, \blacktriangledown) \rightsquigarrow_{\blacktriangledown c_3} (F, y, \pi, E, \blacktriangle)$ **with** $E(y) = \blacktriangledown$.
 1. *Backtracking Code:* by hypothesis $E(y) = \blacktriangledown$, and so $y \in \Lambda(E) =_{L.5.3} \Lambda(F)$.
 2. *Pairs in the Frame:* it follows from the *i.h..*
- **Case** $(y : F, \bar{w}, \epsilon, E, \blacktriangle) \rightsquigarrow_{\blacktriangle c_4} (F, ly.\bar{w}, \epsilon, \blacktriangle y : E, \blacktriangle)$.
 1. *Backtracking Code:* by *i.h.* $\text{fv}(\bar{w}) \subseteq \Lambda(y : F)$ and so $\text{fv}(\lambda y.\bar{w}) = \text{fv}(\bar{w}) \setminus \{x\} = \Lambda(F)$.
 2. *Pairs in the Frame:* it follows from the *i.h..*
- **Case** $((\bar{w}, \pi) : F, \bar{r}, \epsilon, E, \blacktriangle) \rightsquigarrow_{\blacktriangle c_5} (F, \bar{w}\bar{r}, \pi, E, \blacktriangle)$.
 1. *Backtracking Code:* by *i.h.* $\text{fv}(\bar{r}) \subseteq \Lambda((\bar{w}, \pi) : F) = \Lambda(F)$ and by Point 2 of the *i.h.* $\text{fv}(\bar{w}) \subseteq \Lambda(F)$, and so $\text{fv}(\bar{w}\bar{r}) \subseteq \Lambda(F)$.
 2. *Pairs in the Frame:* it follows from the *i.h..*
- **Case** $(F, \bar{w}, \bar{r} : \pi, E, \blacktriangle) \rightsquigarrow_{\blacktriangle c_6} ((\bar{w}, \pi) : F, \bar{r}, \epsilon, E, \blacktriangledown)$.
 1. *Backtracking Code:* nothing to prove.
 2. *Pairs in the Frame:* by Point 1 of the *i.h.* $\text{fv}(\bar{w}) \subseteq \Lambda(F)$, the rest follows from the *i.h..* □

C.5 Proof of the Name Invariant (Lemma 6.4)

Proof. The invariant trivially holds for an initial state $\epsilon \mid \overline{w}_0 \mid \epsilon \mid \epsilon \mid \blacktriangledown$ if \overline{w}_0 is closed and well-named. For a non-empty evaluation sequence we list the cases for the last transitions:

- **Case** $(F, \overline{w}\overline{r}, \pi, E, \blacktriangledown) \rightsquigarrow_{\blacktriangledown c_1} (F, \overline{w}, \overline{r} : \pi, E, \blacktriangledown)$. Every point follows from its *i.h..*
- **Case** $(F, ly.\overline{w}, \overline{r} : \pi, E, \blacktriangledown) \rightsquigarrow_m (F, \overline{w}, \pi, [y \leftarrow \overline{r}] : E, \blacktriangledown)$.
 1. *Substitutions*: for $[y \leftarrow \overline{r}]$ it follows from Point 3 of the *i.h..*, for E it follows from the *i.h..*
 2. *Markers*: note that by Point 3 of the *i.h..* y simply cannot occur in F , the rest follows from the *i.h..*
 3. *Abstractions*: it follows from the *i.h..*
- **Case** $(F, ly.\overline{w}, \epsilon, E, \blacktriangledown) \rightsquigarrow_{\blacktriangledown c_2} (y : F, \overline{w}, \epsilon, \blacktriangledown y : E, \blacktriangledown)$.
 1. *Substitutions*: it follows from the *i.h..*
 2. *Markers*: for y it follows from Point 3 of the *i.h..*, the rest follows from the *i.h..*
 3. *Abstractions*: it follows from the *i.h..*
- **Case** $(F, y, \pi, E, \blacktriangledown) \rightsquigarrow_e (F, \overline{w}^\alpha, \pi, E, \blacktriangledown)$. It follows by the *i.h..* and the fact that in \overline{w}^α the abstracted variables are renamed (wrt \overline{w}) with fresh names.
- **Case** $(F, y, \pi, E, \blacktriangledown) \rightsquigarrow_{\blacktriangledown c_3} (F, y, \pi, E, \blacktriangle)$. Every point follows from its *i.h..*
- **Case** $(y : F, \overline{w}, \epsilon, E, \blacktriangle) \rightsquigarrow_{\blacktriangle c_4} (F, ly.\overline{w}, \epsilon, \blacktriangle y : E, \blacktriangle)$. By the compatibility invariant (Lemma 6.1) $(y : F) \propto E$, and by the factorization property of compatible pairs (Lemma 5.2) $E = E_w : \blacktriangledown y : E'$.
 1. *Substitutions*: it follows from the *i.h..*
 2. *Markers*: it follows from the *i.h..*
 3. *Abstractions*: for $\lambda y. \overline{w}$ it holds because by Point 2 of the *i.h..* y does not appear in F nor in E_t (it may however occur in E_w , but this is taken into account by the statement). For the other abstractions Point 2 follows from the *i.h..*
- **Case** $((\overline{w}, \pi) : F, \overline{r}, \epsilon, E, \blacktriangle) \rightsquigarrow_{\blacktriangle c_5} (F, \overline{w}\overline{r}, \pi, E, \blacktriangle)$. Every point follows from its *i.h..*
- **Case** $(F, \overline{w}, \overline{r} : \pi, E, \blacktriangle) \rightsquigarrow_{\blacktriangle c_6} ((\overline{w}, \pi) : F, \overline{r}, \epsilon, E, \blacktriangledown)$. Every point follows from its *i.h..* \square

C.6 Proof of the Closure Invariant (Lemma 6.5)

Proof. The invariant trivially holds for an initial state $\epsilon \mid \overline{t}_0 \mid \epsilon \mid \epsilon \mid \blacktriangledown$ if \overline{t}_0 is closed and well-named. For a non-empty evaluation sequence we list the cases for the last transitions:

- **Case** $(F, \overline{w}\overline{r}, \pi, E, \blacktriangledown) \rightsquigarrow_{\blacktriangledown c_1} (F, \overline{w}, \overline{r} : \pi, E, \blacktriangledown)$. Every point follows from its *i.h..*
- **Case** $(F, ly.\overline{w}, \overline{r} : \pi, E, \blacktriangledown) \rightsquigarrow_m (F, \overline{w}, \pi, [y \leftarrow \overline{r}] : E, \blacktriangledown)$.
 1. *Environment*: for $[y \leftarrow \overline{r}]$ it follows from Point 2 of the *i.h..*, for the rest it follows from the *i.h..*

2. *Code, Stack, and Frame*: for y is evident, as $[y \leftarrow \bar{r}] : E$ is clearly defined on y , for the rest it follows from the *i.h.*.
- **Case** $(F, ly.\bar{w}, \epsilon, E, \blacktriangledown) \rightsquigarrow_{\blacktriangledown c_2} (y : F, \bar{w}, \epsilon, \blacktriangledown y : E, \blacktriangledown)$.
 1. *Environment*: it follows from the *i.h.*.
 2. *Code, Stack, and Frame*: for y is evident, as $\blacktriangledown y : E$ is clearly defined on y , for the rest it follows from the *i.h..*
 - **Case** $(F, y, \pi, E, \blacktriangledown) \rightsquigarrow_{\pi} (F, \bar{w}^\alpha, \pi, E, \blacktriangledown)$.
 1. *Environment*: it follows from the *i.h..*
 2. *Code, Stack, and Frame*: for \bar{w}^α it follows from Point 1 of the *i.h.*, as \bar{w} appears in the environment out of all closed scopes (otherwise the transition would not take place). The rest follows from the *i.h..*
 - **Case** $(F, y, \pi, E, \blacktriangledown) \rightsquigarrow_{\blacktriangledown c_3} (F, y, \pi, E, \blacktriangle)$ **with** $E(y) = \blacktriangledown$.
 1. *Environment*: it follows from the *i.h..*
 2. *Code, Stack, and Frame*: it follows from the *i.h..*
 - **Case** $(y : F, \bar{w}, \epsilon, E, \blacktriangle) \rightsquigarrow_{\blacktriangle c_4} (F, ly.\bar{w}, \epsilon, \blacktriangle y : E, \blacktriangle)$. By the compatibility invariant (Lemma 6.1) $(y : F) \propto E$, and by the factorization property of compatible pairs (Lemma 5.2) $E = E_w : \blacktriangledown y : E'$.
 1. *Environment*: it follows from the *i.h..*
 2. *Code, Stack, and Frame*: note that
 - (a) E_w does not bind any variable occurring free in \bar{w} by Lemma 6.3.1,
 - (b) E_w does not bind any variable occurring free in F by Lemma 6.4.2, and
 - (c) the stack is empty by hypothesis.
 Then E_w does not bind any free variable in the code, in the stack, nor in the frame, and we conclude using the *i.h..*, because $\blacktriangle x E_w : \blacktriangledown x : E'$ by definition is defined on a variable z iff E' is.
 - **Case** $((\bar{w}, \pi) : F, \bar{r}, \epsilon, E, \blacktriangle) \rightsquigarrow_{\blacktriangle c_5} (F, \bar{w}\bar{r}, \pi, E, \blacktriangle)$.
 1. *Environment*: it follows from the *i.h..*
 2. *Code, Stack, and Frame*: it follows from the *i.h..*
 - **Case** $(F, \bar{w}, \bar{r} : \pi, E, \blacktriangle) \rightsquigarrow_{\blacktriangle c_6} ((\bar{w}, \pi) : F, \bar{r}, \epsilon, E, \blacktriangledown)$.
 1. *Environment*: it follows from the *i.h..*
 2. *Code, Stack, and Frame*: it follows from the *i.h..*

□

D Proofs Omitted from Sect. 6 (Distilling the Strong MAM)

D.1 Proof of Closed Scopes Disappear (Lemma 7)

Proof. Essentially it follows from $\underline{\blacktriangle x : E_w : \blacktriangledown x : E} = \underline{E}$. Precisely, by Lemma 5.2 F and E have, respectively, the forms $F_w : F_t$ and $E'_w : E_t$. Now,

$$\begin{aligned} \underline{F \propto (\blacktriangle x : E_w : \blacktriangledown x : E)} &= \underline{(F_w : F_t) \propto (\blacktriangle x : E_w : \blacktriangledown x : E'_w : E_t)} \\ &= \underline{F_t \propto E_t \langle \blacktriangle x : E_w : \blacktriangledown x : E'_w \langle F_w \rangle \rangle} \\ &= \underline{F_t \propto E_t \langle E'_w \langle F_w \rangle \rangle} \\ &= \underline{(F_w : F_t) \propto (E'_w : E_t)} &&= \underline{F \propto E} \end{aligned}$$

□

D.2 Proof of the Leftmost-Outermost Invariant (Lemma 8)

For the invariant we need the following lemma.

Lemma 12 (Compatible Pairs Decode to Non-Applicative Contexts).

Let F_w be a weak frame, E_w a weak environment, and $F \propto E$ a compatible pair. Then $\underline{F_w}$, $\underline{E_w}$, and $\underline{F \propto E}$ are contexts that are not applicative, i.e. not of the form $C\langle Lt \rangle$.

Proof. The fact that $\underline{F_w}$ and $\underline{E_w}$ are not applicative is an immediate induction over their structure. For $\underline{F \propto E}$ we reason by induction on the compatibility of F and E . The base case $\underline{\epsilon \propto \epsilon} = \langle \cdot \rangle$ is evident. Inductive cases:

1. *Weak Extension*, i.e. $(F_w : F_t) \propto (E_w : E_t)$ with $F_t \propto E_t$. By *i.h.* $\underline{F_t \propto E_t}$ is not applicative and both $\underline{F_w}$ and $\underline{E_w}$ are not applicable. By definition, $(F_w : F_t) \propto (E_w : E_t) = \underline{F_t \propto E_t} \langle \underline{E_w} \langle \underline{F_w} \rangle \rangle$, which is then not applicable.
2. *Abstraction*, i.e. $(x : F) \propto (\nabla x : E)$ with $F \propto E$. Immediate, as $\underline{F \propto E} \langle lx. \langle \cdot \rangle \rangle$ is not applicable. \square

We can now prove that the decoding of the data-structures of a reachable state is a LO context.

Proof (Leftmost-Outermost Invariant, Lemma 8).

We prove that $\underline{F \propto E}$ is a LO context, the fact that C_s is a LO contexts then easily follows, as $C_s := \underline{F \propto E} \langle \underline{\pi} \rangle$.

The invariant trivially holds for an initial state $\epsilon \mid \bar{t}_0 \mid \epsilon \mid \epsilon \mid \nabla$. For a non-empty evaluation sequence we list the cases for the last transitions. We omit the cases for which the environment and the frame do not change (i.e. $\rightsquigarrow_{\nabla c_1}, \rightsquigarrow_e, \rightsquigarrow_{\nabla c_3}$), as for them the statement follows from the *i.h.*.

- **Case** $(F, lx.\bar{t}, \bar{u} : \pi, E, \nabla) \rightsquigarrow_m (F, \bar{t}, \pi, [x \leftarrow \bar{u}] : E, \nabla)$. By *i.h.* $\underline{F \propto E}$ is LO. Let $F = F_w : F_t$, so that $\underline{F \propto E} = \underline{F_t \propto E} \langle \underline{F_w} \rangle$. Note that, by the name invariant (Lemma 6.4.3), the eventual occurrences of x are all in \bar{t} and so $x \notin \text{fv}(F_w)$, and in particular $x \notin \text{1fv}(F_w)$. Then, $\underline{F_t \propto E} \langle \underline{F_w} [x \leftarrow \bar{u}] \rangle$ is LO: the conditions of Definition 6.6 are satisfied either because $\underline{F \propto E} = \underline{F_t \propto E} \langle \underline{F_w} \rangle$ is LO or because $x \notin \text{1fv}(F_w)$.
- **Case** $(F, lx.\bar{t}, \epsilon, E, \nabla) \rightsquigarrow_{\nabla c_2} (x : F, \bar{t}, \epsilon, \nabla x : E, \nabla)$. By *i.h.* we have $\underline{F \propto E}$ is LO and by Lemma 12 $\underline{F \propto E}$ is not applicative, so $(x : F) \propto (\nabla x : E) = \underline{F \propto E} \langle lx. \langle \cdot \rangle \rangle$ is LO (it satisfies the conditions of Definition 6.6 because $\underline{F \propto E}$ does).
- **Case** $(x : F, \bar{t}, \epsilon, E, \blacktriangle) \rightsquigarrow_{\blacktriangle c_4} (F, lx.\bar{t}, \epsilon, \blacktriangle x : E, \blacktriangle)$. By the compatibility invariant (Lemma 6.1) $(x : F) \propto E$, and by the factorization property of compatible pairs (Lemma 5.2) $E = E_w : \nabla x : E'$. By definition

$$(x : F) \propto (E_w : \nabla x : E_t) = \underline{F \propto E_t} \langle lx. \underline{E_w} \rangle$$

that by *i.h.* is LO. Now, $\underline{F \propto E_t}$ is LO, as it satisfies the conditions of Definition 6.6 because $\underline{F \propto E}$ does. We conclude by noticing that the compatible pair of the target state satisfies $\underline{F \propto (\blacktriangle x : E)} = \underline{F \propto (\blacktriangle x : E_w : \nabla x : E_t)} =_{L.7} \underline{F \propto E_t}$.

- **Case** $((\bar{t}, \pi) : F, \bar{u}, \epsilon, E, \blacktriangle) \rightsquigarrow_{\textcolor{red}{c}_5} ((F, \bar{t}\bar{u}, \pi, E, \blacktriangle))$. By *i.h.* we have that $((\bar{t}, \pi) : F) \propto E$ is LO and by *frame* part of the backtracking normal form invariant (Lemma 6.3.2) \bar{t} is neutral. By definition, $((\bar{t}, \pi) : F) \propto E = \underline{F} \propto \underline{E} \langle \underline{\pi} \langle \bar{t} \cdot \rangle \rangle$, Then, $\underline{F} \propto \underline{E}$ —being a prefix of $((\bar{t}, \pi) : F) \propto E$ —verifies the conditions of Definition 6.6 and is LO.
- **Case** $(F, \bar{t}, \bar{u} : \pi, E, \blacktriangle) \rightsquigarrow_{\textcolor{red}{c}_6} ((\bar{t}, \pi) : F, \bar{u}, \epsilon, E, \blacktriangledown)$. Note that
 1. $\underline{F} \propto \underline{E}$ is LO by *i.h.*,
 2. $\underline{F} \propto \underline{E}$ is not applicative by Lemma 12,
 3. $\text{fv}(\bar{t}) \subseteq \Lambda(F)$ by the backtracking free variables invariant (Lemma 6.3.1).
 4. \bar{t} is a neutral term by the normal form invariant (Lemma 6.2.1), because the stack at the left-hand side is not empty.
 Note that Point 3 guarantees that $x \notin \text{fv}(\bar{t})$, and so in particular $x \notin \text{fv}(\bar{t})$, for any ES $[x \leftarrow \bar{w}]$ in E (and so in $\underline{F} \propto \underline{E}$). Then $\underline{F} \propto \underline{E} \langle \underline{\pi} \langle \bar{t} \cdot \rangle \rangle$ is LO (because it verifies the conditions of Definition 6.6, by the listed points), that is to say $((\bar{t}, \pi) : F) \propto E$ is LO. \square

D.3 Proof of the Properties of the Decoding wrt Structural Equivalence \equiv (Lemma 9)

We here present a more general statement than the one in the paper. The reason is that the proof of the second point of the lemma (*Compatible Pairs Absorb Substitutions*) actually requires a further lemma (*Weak Frames and Substitutions Commute* below) that is omitted from the statement in the paper because it is not used anywhere else.

Lemma 13 (Decoding and Structural Equivalence \equiv).

1. Stacks and Substitutions Commute: if x does not occur free in π then $\underline{\pi} \langle t[x \leftarrow u] \rangle \equiv \underline{\pi} \langle t \rangle [x \leftarrow u]$;
2. Weak Frames and Substitutions Commute: if x does not occur free in F_w then $\underline{F_w} \langle t[x \leftarrow u] \rangle \equiv \underline{F_w} \langle t \rangle [x \leftarrow u]$;
3. Compatible Pairs Absorb Substitutions: if x does not occur free in F then $\underline{F} \propto \underline{E} \langle t[x \leftarrow u] \rangle \equiv \underline{F} \propto ([x \leftarrow u] : E) \langle t \rangle$.

Proof.

1. *Stacks and Substitutions Commute:* by induction on π . Cases:
 - (a) *Empty Stack*, i.e. $\pi = \epsilon$. Then $\underline{\epsilon} \langle t[x \leftarrow u] \rangle = t[x \leftarrow u] = \underline{\epsilon} \langle t \rangle [x \leftarrow u]$.
 - (b) *Non-Empty Stack*, i.e. $\pi = \bar{w} : \pi'$. Then

$$\begin{aligned} \underline{\bar{w}} : \underline{\pi'} \langle t[x \leftarrow u] \rangle &= \underline{\pi'} \langle t[x \leftarrow u] \rangle \bar{w} \\ &\equiv_{\text{i.h.}} \underline{\pi'} \langle t \rangle [x \leftarrow u] \bar{w} \\ &\equiv_{\text{@r}} \underline{\pi'} \langle t \rangle \bar{w}[x \leftarrow u] = \underline{\bar{w}} : \underline{\pi'} \langle t \rangle [x \leftarrow u] \end{aligned}$$

Note that the proof uses only $\equiv_{\text{@1}}$.

2. *Weak Frames and Substitutions Commute:* by induction on F_w . Cases:
 - (a) *Empty Weak Frame*, i.e. $F_w = \epsilon$. Then $\underline{\epsilon} \langle t[x \leftarrow u] \rangle = t[x \leftarrow u] = \underline{\epsilon} \langle t \rangle [x \leftarrow u]$.

(b) *Non-Empty Weak Frame*, i.e. $F_w = (\bar{w}, \pi) : F'_w$. Then

$$\begin{aligned} \underline{(\bar{w}, \pi) : F'_w \langle t[x \leftarrow u] \rangle} &= \underline{\frac{F'_w \langle \underline{\pi} \langle \bar{w}(t[x \leftarrow u]) \rangle \rangle}{\equiv_{@r} \frac{F'_w \langle \underline{\pi} \langle (\bar{w}t)[x \leftarrow u] \rangle \rangle}{\equiv_{P.1} \frac{F'_w \langle \underline{\pi} \langle (\bar{w}t) \rangle [x \leftarrow u] \rangle}{\equiv_{i.h.} \underline{F'_w \langle \underline{\pi} \langle \bar{w}t \rangle \rangle [x \leftarrow u]} = (\bar{w}, \pi) : F'_w \langle t \rangle [x \leftarrow u]}}}}} \end{aligned}$$

Note that the proof uses only $\equiv_{@r}$ and $\equiv_{@1}$ (because of the previous point).

3. *Compatible Pairs Absorb Substitutions*: By Lemma 5.2 we can decompose F and E in their weak and trunk parts, obtaining:

$$\begin{aligned} \underline{F \propto E \langle t[x \leftarrow u] \rangle} &= \underline{(F_w : F_t) \propto (E_w : E_t \langle t[x \leftarrow u] \rangle)} \\ &= \underline{\frac{F_t \propto E_t \langle E_w \langle F_w \langle t[x \leftarrow u] \rangle \rangle \rangle}{=_{P.2} \frac{F_t \propto E_t \langle E_w \langle F_w \langle t \rangle [x \leftarrow u] \rangle \rangle}{= \frac{F_t \propto E_t \langle [x \leftarrow u] : E_w \langle F_w \langle t \rangle \rangle \rangle}{= \frac{(F_w : F_t) \propto ([x \leftarrow u] : E_w : E_t) \langle t \rangle}{= F \propto ([x \leftarrow u] : E) \langle t \rangle}}} \quad \square} \end{aligned}$$

D.4 Cases Omitted from the Proof of the Distillation Theorem (Theorem 3)

Proof. We list here the equality cases omitted from the main proof in the paper.

- **Case** $(F, \bar{t}\bar{u}, \pi, E, \blacktriangledown) \rightsquigarrow_{\blacktriangledown c_1} (F, \bar{t}, \bar{u} : \pi, E, \blacktriangledown)$.

$$\underline{(F, \bar{t}\bar{u}, \pi, E, \blacktriangledown)} = \underline{F \propto E \langle \underline{\pi} \langle \bar{t}\bar{u} \rangle \rangle} = \underline{F \propto E \langle \bar{u} : \pi \langle \bar{t} \rangle \rangle} = \underline{(F, \bar{t}, \bar{u} : \pi, E, \blacktriangledown)}$$

- **Case** $(F, lx.\bar{t}, \epsilon, E, \blacktriangledown) \rightsquigarrow_{\blacktriangledown c_2} (x : F, \bar{t}, \epsilon, \blacktriangledown x : E, \blacktriangledown)$.

$$\begin{aligned} \underline{(F, lx.\bar{t}, \epsilon, E, \blacktriangledown)} &= \underline{F \propto E \langle lx.\bar{t} \rangle} \\ &= \underline{(x : F) \propto (\blacktriangledown x : E) \langle \bar{t} \rangle} = \underline{(x : F, \bar{t}, \epsilon, \blacktriangledown x : E, \blacktriangledown)} \end{aligned}$$

- **Case** $(F, x, \pi, E, \blacktriangledown) \rightsquigarrow_{\blacktriangledown c_3} (F, x, \pi, E, \blacktriangle)$.

$$\underline{(F, x, \pi, E, \blacktriangledown)} = \underline{F \propto E \langle \underline{\pi} \langle x \rangle \rangle} = \underline{(F, x, \pi, E, \blacktriangle)}$$

- **Case** $((\bar{t}, \pi) : F, \bar{u}, \epsilon, E, \blacktriangle) \rightsquigarrow_{\blacktriangle c_5} (F, \bar{t}\bar{u}, \pi, E, \blacktriangle)$.

$$\underline{((\bar{t}, \pi) : F, \bar{u}, \epsilon, E, \blacktriangle)} = \underline{(\bar{t}, \pi) : F \propto E \langle \bar{u} \rangle} = \underline{F \propto E \langle \pi \langle \bar{t} \bar{u} \rangle \rangle} = \underline{(F, \bar{t}\bar{u}, \pi, E, \blacktriangle)}$$

- **Case** $(F, \bar{t}, \bar{u} : \pi, E, \blacktriangle) \rightsquigarrow_{\blacktriangle c_6} ((\bar{t}, \pi) : F, \bar{u}, \epsilon, E, \blacktriangledown)$.

$$\begin{aligned} \underline{(F, \bar{t}, \bar{u} : \pi, E, \blacktriangle)} &= \underline{F \propto E \langle \bar{u} : \pi \langle \bar{t} \rangle \rangle} \\ &= \underline{F \propto E \langle \underline{\pi} \langle \bar{t} \bar{u} \rangle \rangle} \\ &= \underline{((\bar{t}, \pi) : F) \propto E \langle \bar{u} \rangle} = \underline{((\bar{t}, \pi) : F, \bar{u}, \epsilon, E, \blacktriangledown)} \end{aligned}$$

□

E Proofs Omitted from Sect. 7 (Complexity Analysis)

The proof of the subterm invariant (Lemma 10) for the machine is in the next subsection, and it is obtained as a corollary of a more general invariant. The subterm property for \rightarrow_{LO} (Lemma 11) is an immediate consequence of Lemma 10 and the case of exponential transition in the distillation theorem (Theorem 3).

E.1 Proof of the Subterm Invariant (Lemma 10)

The subterm invariant as formulated in the paper is a consequence of the last point of the following more general invariant, because $\rightsquigarrow_{\text{E}}$ duplicates codes from the environment, here proved to be subterms of the initial term.

Lemma 14 (Subterm Invariant). *Let $s = F \mid \bar{u} \mid \pi \mid E \mid \varphi$ be a state reachable from the initial code \bar{t} . Then*

1. Evaluating Code: if $\varphi = \blacktriangledown$, then \bar{u} is a subterm of \bar{t} ;
2. Stack: any code in the stack π is a subterm of \bar{t} ;
3. Frame: if $F = F' : (\bar{w}, \pi') : F''$, then any code in π' is a subterm of \bar{t} ;
4. Global Environment: if $E = E' : [x \leftarrow \bar{w}] : E''$, then \bar{w} is a subterm of \bar{t} ;

Proof. Let us use \bar{t}_0 for the initial term. The invariant trivially holds for the initial state $\epsilon \mid \bar{t}_0 \mid \epsilon \mid \epsilon \mid \blacktriangledown$. In the inductive case we look at the last transition:

- **Case** $(F, \bar{t}\bar{u}, \pi, E, \blacktriangledown) \rightsquigarrow_{\text{Vc}_1} (F, \bar{t}, \bar{u} : \pi, E, \blacktriangledown)$.
 1. *Evaluating Code:* By i.h., $\bar{t}\bar{u}$ is a subterm of \bar{t}_0 , so \bar{t} is also a subterm of \bar{t}_0 .
 2. *Stack:* by i.h., $\bar{t}\bar{u}$ is a subterm of \bar{t}_0 , so \bar{u} is also a subterm of \bar{t}_0 . Moreover, any piece of code in π is a subterm of \bar{t}_0 by i.h..
 3. *Frame:* it follows from the i.h., since the frame F is unchanged.
 4. *Environment:* it follows from the i.h., since the environment E is unchanged.
- **Case** $(F, lx.\bar{t}, \bar{u} : \pi, E, \blacktriangledown) \rightsquigarrow_{\text{m}} (F, \bar{t}, \pi, [x \leftarrow \bar{u}] : E, \blacktriangledown)$.
 1. *Evaluating Code:* note that \bar{t} is a subterm of $lx.\bar{t}$.
 2. *Stack:* note that any piece code in π is also in $\bar{u} : \pi$.
 3. *Frame:* it follows from the i.h., since F is not modified.
 4. *Environment:* the new environment is of the form $[x \leftarrow \bar{u}] : E$. Pieces of code in E are subterms of \bar{t}_0 by i.h.. Moreover \bar{u} is the top of the stack $\bar{u} : \pi$ so it is also a subterm of \bar{t}_0 .
- **Case** $(F, lx.\bar{t}, \epsilon, E, \blacktriangledown) \rightsquigarrow_{\text{Vc}_2} (x : F, \bar{t}, \epsilon, \blacktriangledown x : E, \blacktriangledown)$.
 1. *Evaluating Code:* note that \bar{t} is a subterm of $lx.\bar{t}$ which is in turn a subterm of \bar{t}_0 by i.h..
 2. *Stack:* trivial since the stack π is empty.
 3. *Frame:* any pair of the form (\bar{u}, π') in the frame $x : F$ is also already present in F , so by i.h. any piece of code in π' is a subterm of \bar{t}_0 .

4. *Environment*: it follows from the *i.h.*, since the environment E is unchanged.
- **Case** $(F, x, \pi, E, \blacktriangledown) \rightsquigarrow_e (F, \bar{t}^\alpha, \pi, E, \blacktriangledown)$.
 1. *Evaluating Code*: note that \bar{t} is bound by E . By *i.h.*, it is a subterm of \bar{t}_0 . So \bar{t}^α is also a subterm of \bar{t}_0 .
 2. *Stack*: it follows from the *i.h.*, since the stack π is unchanged.
 3. *Frame*: it follows from the *i.h.*, since the frame F is unchanged.
 4. *Environment*: it follows from the *i.h.*, since the environment E is unchanged.
 - **Case** $(F, x, \pi, E, \blacktriangledown) \rightsquigarrow_{\blacktriangledown c_3} (F, x, \pi, E, \blacktriangle)$.
 1. *Evaluating Code*: trivial since $\varphi \neq \blacktriangledown$.
 2. *Stack*: it follows from the *i.h.*, since the stack π is unchanged.
 3. *Frame*: it follows from the *i.h.*, since the frame F is unchanged.
 4. *Environment*: it follows from the *i.h.*, since the environment E is unchanged.
 - **Case** $(x : F, \bar{t}, \epsilon, E, \blacktriangle) \rightsquigarrow_{\blacktriangle c_4} (F, lx.\bar{t}, \epsilon, \blacktriangle x : E, \blacktriangle)$.
 1. *Evaluating Code*: trivial since $\varphi \neq \blacktriangledown$.
 2. *Stack*: trivial since the stack is empty.
 3. *Frame*: any pair of the form (\bar{u}, π) in the frame F is also in the frame $x : F$, so any piece of code in π is a subterm of \bar{t}_0 by *i.h.*.
 4. *Environment*: any substitution of the form $[y \leftarrow \bar{u}]$ in the environment $\blacktriangle x : E$ is also in the environment E , so \bar{u} is a subterm of \bar{t}_0 by *i.h.*.
 - **Case** $((\bar{t}, \pi) : F, \bar{u}, \epsilon, E, \blacktriangle) \rightsquigarrow_{\blacktriangle c_5} (F, \bar{t}\bar{u}, \pi, E, \blacktriangle)$.
 1. *Evaluating Code*: trivial since $\varphi \neq \blacktriangledown$.
 2. *Stack*: the stack π occurs at the left-hand side in the frame $(\bar{t}, \pi) : F$, so by *i.h.* we know that any piece of code in π is a subterm of \bar{t}_0 .
 3. *Frame*: any pair (\bar{w}, π) in the frame F is also in the frame $(\bar{t}, \pi) : F$, so any piece of code in π must be a subterm of \bar{t}_0 .
 4. *Environment*: it follows from the *i.h.*, since the environment E is unchanged.
 - **Case** $(F, \bar{t}, \bar{u} : \pi, E, \blacktriangle) \rightsquigarrow_{\blacktriangle c_6} ((\bar{t}, \pi) : F, \bar{u}, \epsilon, E, \blacktriangledown)$.
 1. *Evaluating Code*: note that \bar{u} is an element of the stack at the left-hand side of the transition, so by *i.h.* \bar{u} is a subterm of \bar{t}_0 .
 2. *Stack*: trivial since the stack is empty.
 3. *Frame*: any pair in the frame $(\bar{t}, \pi) : F$ is also in the frame F except for (\bar{t}, π) . Consider a piece of code \bar{r} in the stack π . It is trivially also a piece of code in the stack $\bar{u} : \pi$, so by *i.h.* we have that \bar{r} is a subterm of \bar{t}_0 .
 4. *Environment*: it follows from the *i.h.*, since the environment E is unchanged. \square

F Proof that Structural equivalence is a Strong Bisimulation (Proposition 1)

We first need an auxiliary lemma (Lemma 16), which uses an alternative, inductive definition of LO contexts:

Definition 12 (iLO Contexts). A context C is inductively LO (or iLO) if a judgment about it can be derived using the following inductive rules:

$$\begin{array}{c} \frac{}{\langle \cdot \rangle \text{ is iLO}} (\text{ax-iLO}) \quad \frac{C \text{ is iLO} \quad C \neq L\langle \lambda x. C' \rangle}{Ct \text{ is iLO}} (@l\text{-iLO}) \\[10pt] \frac{C \text{ is iLO}}{\lambda x. C \text{ is iLO}} (l\text{-iLO}) \quad \frac{t \text{ is neutral} \quad C \text{ is iLO}}{tC \text{ is iLO}} (@r\text{-iLO}) \\[10pt] \frac{C \text{ is iLO} \quad x \notin \text{fv}(C)}{C[x \leftarrow t] \text{ is iLO}} (ES\text{-iLO}) \end{array}$$

Lemma 15. A context C is iLO iff it is LO.

Proof. An immediate induction on C . \square

Lemma 16. If C is a LO context and C does not bind any of the variables in $\text{fv}(u)$, then $C\langle t[x \leftarrow u] \rangle \equiv C\langle t \rangle [x \leftarrow u]$.

Proof. A context is LO iff it is iLO (Lemma 15). The property is then proved by induction on the derivation that C is an iLO context. \square

Proof (Structural Equivalence \equiv is a Strong Bisimulation, Proposition 1).

Let \Leftrightarrow be the symmetric and contextual closure of the axioms by which \equiv is defined, i.e.

$$\begin{array}{ll} t[x \leftarrow u] \equiv_{\text{gc}} t & \text{if } x \notin \text{fv}(t) \\ t[x \leftarrow u][y \leftarrow w] \equiv_{\text{com}} t[y \leftarrow w][x \leftarrow u] & \text{if } y \notin \text{fv}(u) \text{ and } x \notin \text{fv}(w) \\ t[x \leftarrow u][y \leftarrow w] \equiv_{[]} t[x \leftarrow u[y \leftarrow w]] & \text{if } y \notin \text{fv}(t) \\ t[x \leftarrow u] \equiv_{\text{dup}} t_{[y]_x}[x \leftarrow u][y \leftarrow u] \\ (lx.t)[y \leftarrow u] \equiv_{\lambda} lx.t[y \leftarrow u] & \text{if } x \notin \text{fv}(u) \\ (tu)[x \leftarrow w] \equiv_{@1} t[x \leftarrow w]u & \text{if } x \notin \text{fv}(u) \\ (tu)[x \leftarrow w] \equiv_{@r} tu[x \leftarrow w] & \text{if } x \notin \text{fv}(t) \end{array}$$

Note that \equiv is the reflexive-transitive closure of \Leftrightarrow . It suffices to show that $\Leftrightarrow \rightarrow_{\text{LO}} \subseteq \rightarrow_{\text{LO}} \equiv$, preserving the kind of step (multiplicative/exponential). The fact that \Leftrightarrow^* is a bisimulation then follows by induction on the number of \Leftrightarrow steps.

Let $w \Leftrightarrow t \rightarrow_{\text{LO}} u$. The proof of $w \rightarrow_{\text{LO}} \equiv u$ goes by induction on the context under which the step $t \rightarrow_{\text{LO}} u$ takes place. In the following proof note that:

1. \rightarrow_m steps are sent to \rightarrow_m steps,
2. \rightarrow_e steps are sent to \rightarrow_e steps, and
3. no step is ever duplicated.

Cases:

1. **Base case 1: multiplicative root step**, $t = L\langle lx.t' \rangle u' \mapsto_m L\langle t'[x \leftarrow u'] \rangle$. If the \Leftrightarrow step is internal to t' , internal to u' , or internal to the argument of one of the substitutions in L , then the pattern of the \Leftrightarrow redex does not overlap with the \mapsto_m step, and the proof is immediate, as the two steps commute. Otherwise, we consider every possible case of \Leftrightarrow :

- (a) *Garbage collection*, \equiv_{gc} . The garbage collected substitution must be one of the substitutions in L , i.e. L must be of the form $L'\langle L''[y \leftarrow w'] \rangle$. Then:

$$\begin{aligned} L'\langle L''\langle lx.t' \rangle [y \leftarrow z] \rangle u' &\xrightarrow{\text{m}} L'\langle L''\langle t'[x \leftarrow u'] \rangle [y \leftarrow z] \rangle \\ &\stackrel{\equiv_{gc}}{\equiv} L'\langle L''\langle lx.t' \rangle \rangle u' \xrightarrow{\text{m}} L'\langle L''\langle t'[x \leftarrow u'] \rangle \rangle \end{aligned}$$

- (b) *Commutation of independent substitutions*, \equiv_{com} . The substitutions that are commuted must be both in L , i.e. L must be of the form $L'\langle L''[y \leftarrow w'][z \leftarrow r'] \rangle$. Then:

$$\begin{aligned} L'\langle L''\langle lx.t' \rangle [y \leftarrow w'][z \leftarrow r'] \rangle u' &\xrightarrow{\text{m}} L'\langle L''\langle t'[x \leftarrow u'] \rangle [y \leftarrow w'][z \leftarrow r'] \rangle \\ &\stackrel{\equiv_{com}}{\equiv} L'\langle L''\langle lx.t' \rangle [z \leftarrow r'][y \leftarrow w'] \rangle u' \xrightarrow{\text{m}} L'\langle L''\langle t'[x \leftarrow u'] \rangle [z \leftarrow r'][y \leftarrow w'] \rangle \end{aligned}$$

- (c) *Composition of substitutions*, $\equiv_{[\cdot]}$. The substitutions that are composed must be both in L , i.e. L must be of the form $L'\langle L''[y \leftarrow w'][z \leftarrow r'] \rangle$. Then:

$$\begin{aligned} L'\langle L''\langle lx.t' \rangle [y \leftarrow w'][z \leftarrow r'] \rangle u' &\xrightarrow{\text{m}} L'\langle L''\langle t'[x \leftarrow u'] \rangle [y \leftarrow w'][z \leftarrow r'] \rangle \\ &\stackrel{\equiv_{[\cdot]}}{\equiv} L'\langle L''\langle lx.t' \rangle [y \leftarrow w'[z \leftarrow r']] \rangle u' \xrightarrow{\text{m}} L'\langle L''\langle t'[x \leftarrow u'] \rangle [y \leftarrow w'[z \leftarrow r']] \rangle \end{aligned}$$

- (d) *Duplication*, \equiv_{dup} . The duplicated substitution must be one of the substitutions in L , i.e. L must be of the form $L'\langle L''[y \leftarrow w'] \rangle$. Then:

$$\begin{aligned} L'\langle L''\langle lx.t' \rangle [y \leftarrow w'] \rangle u' &\xrightarrow{\text{m}} L'\langle L''\langle t'[x \leftarrow u'] \rangle [y \leftarrow w'] \rangle \\ &\stackrel{\equiv_{dup}}{\equiv} L'\langle (L''\langle lx.t' \rangle)_{[z]_y} [y \leftarrow w'][z \leftarrow w'] \rangle u' \xrightarrow{\text{m}} L'\langle (L''\langle t'[x \leftarrow u'] \rangle)_{[z]_y} [y \leftarrow w'][z \leftarrow w'] \rangle \end{aligned}$$

- (e) *Commutation with abstraction*, \equiv_λ . The commuted substitution must be the innermost substitution in L , i.e. L must be of the form $L'\langle [y \leftarrow w'] \rangle$, and:

$$\begin{aligned} L'\langle (lx.t')[y \leftarrow w'] \rangle u' &\xrightarrow{\circ} L'\langle t'[x \leftarrow u'][y \leftarrow w'] \rangle \\ &\stackrel{\equiv_\lambda}{\equiv} L'\langle lx.t'[y \leftarrow w'] \rangle u' \xrightarrow{\circ} L'\langle t'[y \leftarrow w'][x \leftarrow u'] \rangle \end{aligned}$$

Note that the diagram can be also read from the bottom-up for a reverse application of the \equiv_λ rule. In order to be able to apply \equiv_{com} , note that $x \notin \text{fv}(w')$ by application of the \equiv_λ rule, and that $y \notin \text{fv}(u')$ by the bound variable convention.

- (f) *Left commutation with application*, $\equiv_{@1}$. The only possibility is that the outermost substitution of L commutes with the application taking part in the \rightarrow_m step. That is, L must be of the form $L'[y \leftarrow w']$ and:

$$\begin{aligned} L' \langle lx.t' \rangle [y \leftarrow w'] u' &\xrightarrow{\equiv} L' \langle t'[x \leftarrow u'] \rangle [y \leftarrow w'] u' \\ &\stackrel{\equiv_{@1}}{=} \\ (L' \langle lx.t' \rangle u') [y \leftarrow w'] &\dashrightarrow^m L' \langle t'[x \leftarrow u'] \rangle [y \leftarrow w'] \end{aligned}$$

- (g) *Right commutation with application*, $\equiv_{@r}$. Note that every $\equiv_{@r}$ (and $\equiv_{@r}^{-1}$) redex in $(lx.t')L u'$ must be internal to either t' , u' , or the argument of one of the substitutions in L . We have already argued that in these cases the steps commute.

2. **Base case 2: exponential root step**, $t = C \langle x \rangle [x \leftarrow t'] \mapsto_e C \langle t' \rangle [x \leftarrow t']$. If the substitution that is contracted by the exponential step does not take part in the pattern of the \Leftrightarrow step, it is immediate to check that the property holds. More precisely, suppose that $C \langle x \rangle [x \leftarrow t'] \Leftrightarrow C' \langle x \rangle [x \leftarrow t'']$, where C' and t'' result respectively from C and t by a single step of \Leftrightarrow . Note that we have that either $C \Leftrightarrow C'$ and $t' = t''$ or vice-versa. Then:

$$\begin{aligned} C \langle x \rangle [x \leftarrow t'] &\xrightarrow{\circ} C \langle t' \rangle [x \leftarrow t'] \\ \Leftrightarrow \\ C' \langle x \rangle [x \leftarrow t''] &\dashrightarrow^e C' \langle t'' \rangle [x \leftarrow t''] \end{aligned}$$

Note that when commutation affects t' (*i.e.* if we are in the case in which $C = C'$ and $t' \Leftrightarrow t''$), then the right-hand side of the diagram must be closed by two \Leftrightarrow steps: one for each copy of t' .

So we may assume that the substitution that is contracted by the exponential step does take part in the pattern of the \Leftrightarrow step. We consider every possible case of \Leftrightarrow .

- (a) *Garbage collection*, \equiv_{gc} . The garbage collected substitution cannot erase the contracted occurrence of x , since C is a LO context, and it cannot go inside substitutions. Two subcases, depending on the position of the hole of C with respect to the node of the garbage collected substitution:
i. If the hole of C lies inside the body of the garbage collected substitution, *i.e.* $C = C' \langle C''[y \leftarrow u'] \rangle$ with $y \notin \text{fv}(C'' \langle x \rangle)$, then:

$$\begin{aligned} C' \langle C'' \langle x \rangle [y \leftarrow u'] \rangle [x \leftarrow t'] &\xrightarrow{\circ} C' \langle C'' \langle t' \rangle [y \leftarrow u'] \rangle [x \leftarrow t'] \\ &\stackrel{\equiv_{gc}}{=} \\ C' \langle C'' \langle x \rangle \rangle [x \leftarrow t'] &\dashrightarrow^e C' \langle C'' \langle t' \rangle \rangle [x \leftarrow t'] \end{aligned}$$

Note that $y \notin \text{fv}(C'' \langle t' \rangle)$ since we may assume that $y \notin \text{fv}(t')$ by the bound variable convention.

- ii. Otherwise, the hole of C must be disjoint from the node of the garbage collected substitution, *i.e.* there must be a two-hole context C' such that:

$$C = C' \langle \langle \cdot \rangle, u'[y \leftarrow w'] \rangle$$

where $y \notin \text{fv}(u')$. Then:

$$\begin{aligned} C' \langle x, u'[y \leftarrow w'] \rangle [x \leftarrow t'] &\xrightarrow{\circ} C' \langle t', u'[y \leftarrow w'] \rangle [x \leftarrow t'] \\ &\xrightarrow{\equiv_{\text{gc}}} C' \langle x, u' \rangle [x \leftarrow t'] & C' \langle t', u'[y \leftarrow w'] \rangle [x \leftarrow t'] \\ &\xrightarrow{\circ} C' \langle t', u' \rangle [x \leftarrow t'] \end{aligned}$$

- (b) *Commutation of independent substitutions*, \equiv_{com} . Note that the contracted occurrence of x cannot be inside the argument of any of the commuted substitutions, since C is a LO context and it cannot go inside substitutions. Since the contracted substitution is commuted, we have that C must be of the form $C'[y \leftarrow u']$ and the situation is:

$$\begin{aligned} C' \langle x \rangle [y \leftarrow u'] [x \leftarrow t'] &\xrightarrow{\circ} C' \langle t' \rangle [y \leftarrow u'] [x \leftarrow t'] \\ &\xrightarrow{\equiv_{\text{com}}} C' \langle x \rangle [x \leftarrow t'] [y \leftarrow u'] & C' \langle t' \rangle [x \leftarrow t'] [y \leftarrow u'] \\ &\xrightarrow{\circ} C' \langle t' \rangle [x \leftarrow t'] [y \leftarrow u'] \end{aligned}$$

- (c) *Composition of substitutions*, $\equiv_{[\cdot]}$. Note that the contracted occurrence of x cannot be inside the argument of any of the two substitutions that take part in the $\equiv_{[\cdot]}$ step, since C is a LO context and it cannot go inside substitutions. We know that the contracted substitution takes part in the $\equiv_{[\cdot]}$ step. We consider two subcases, depending on whether the $\equiv_{[\cdot]}$ rule is applied from left to right or from right to left, since the situation is not symmetrical.

- i. If the $\equiv_{[\cdot]}$ step is applied from left to right, then C must be of the form $C'[y \leftarrow u']$ with $x \notin \text{fv}(C' \langle x \rangle)$. This is a contradiction, so this case is not actually possible.
- ii. If the $\equiv_{[\cdot]}$ step is applied from right to left, then t' must be of the form $t''[y \leftarrow u']$ and:

$$\begin{aligned} C \langle x \rangle [x \leftarrow t''[y \leftarrow u']] &\xrightarrow{\circ} C \langle t''[y \leftarrow u'] \rangle [x \leftarrow t''[y \leftarrow u']] \\ &\xrightarrow{\equiv_{[\cdot]}} C \langle x \rangle [x \leftarrow t''[y \leftarrow u']] & C \langle t''[y \leftarrow u'] \rangle [x \leftarrow t''[y \leftarrow u']] \\ &\xrightarrow{\circ} C \langle t''[y \leftarrow u'] \rangle [x \leftarrow t''[y \leftarrow u']] \end{aligned}$$

To close the right-hand side of the diagram, we are left to show that:

$$C \langle t''[y \leftarrow u'] \rangle [x \leftarrow t''[y \leftarrow u']] \equiv C \langle t'' \rangle [x \leftarrow t''[y \leftarrow u']]$$

First note that C is a LO context, and that, by the bound variable convention, C does not bind any of the variables in $\text{fv}(u')$. By resorting to Lemma 16, this allows us to commute the substitution

that:

$$\begin{aligned}
& C\langle t''[y \leftarrow u'] \rangle [x \leftarrow t''[y \leftarrow u']] \\
\equiv & C\langle t'' \rangle [y \leftarrow u'] [x \leftarrow t''[y \leftarrow u']] \quad \text{by Lemma 16} \\
\equiv_{[]} & C\langle t'' \rangle [y \leftarrow u'] [x \leftarrow t''] [y \leftarrow u'] \\
= & C\langle t'' \rangle [y \leftarrow u'] [x \leftarrow t''[y \leftarrow z]] [z \leftarrow u'] \quad \text{renaming } y \text{ to } z \\
\equiv_{\text{com}} & C\langle t'' \rangle [x \leftarrow t''[y \leftarrow z]] [y \leftarrow u'] [z \leftarrow u'] \\
\equiv_{\text{dup}} & C\langle t'' \rangle [x \leftarrow t''] [y \leftarrow u']
\end{aligned}$$

- (d) *Duplication*, \equiv_{dup} . Note that the contracted occurrence of x cannot be inside the argument of any of the two substitutions that take part in the \equiv_{dup} step, since C is a LO context and it cannot go inside substitutions. We consider two cases, depending on whether \equiv_{dup} is applied from left to right or from right to left:

- i. From left to right: the contracted occurrence of x is either renamed to y or left untouched as x . Let z denote x or y , correspondingly. In both cases we have:

$$\begin{aligned}
C\langle x \rangle [x \leftarrow t'] & \xrightarrow{\circ} C\langle t' \rangle [x \leftarrow t'] \\
\equiv_{\text{dup}} & \\
C_{[y]_x} \langle z \rangle [x \leftarrow t'] [y \leftarrow t'] & \dashv \xrightarrow{\circ} C_{[y]_x} \langle t' \rangle [x \leftarrow t'] [y \leftarrow t']
\end{aligned}$$

- ii. From right to left: then C is of the form $C'_{[x]_y} [y \leftarrow t']$, where C' has no occurrences of x , and:

$$\begin{aligned}
C'_{[x]_y} \langle x \rangle [y \leftarrow t'] [x \leftarrow t'] & \xrightarrow{\circ} C'_{[x]_y} \langle t' \rangle [y \leftarrow t'] [x \leftarrow t'] \\
\equiv_{\text{dup}} & \\
C' \langle y \rangle [y \leftarrow t'] & \dashv \xrightarrow{\circ} C' \langle t' \rangle [y \leftarrow t']
\end{aligned}$$

- (e) *Commutation with abstraction*, \equiv_{λ} . Then C is of the form $ly.C'$ and:

$$\begin{aligned}
(l y. C' \langle x \rangle) [x \leftarrow t'] & \xrightarrow{\circ} (l y. C' \langle t' \rangle) [x \leftarrow t'] \\
\equiv_{\lambda} & \\
l y. C' \langle x \rangle [x \leftarrow t'] & \dashv \xrightarrow{\circ} l y. C' \langle t' \rangle [x \leftarrow t']
\end{aligned}$$

- (f) *Left commutation with application*, $\equiv_{@1}$. Then C is of the form $C u'$ and:

$$\begin{aligned}
(C \langle x \rangle u') [x \leftarrow t'] & \xrightarrow{\circ} (C \langle t' \rangle u') [x \leftarrow t'] \\
\equiv_{@1} & \\
C \langle x \rangle [x \leftarrow t'] u' & \dashv \xrightarrow{\circ} C \langle t' \rangle [x \leftarrow t'] u'
\end{aligned}$$

- (g) *Right commutation with application*, $\equiv_{@r}$. Then C is of the form $u' C$ and:

$$\begin{aligned}
(u' C \langle x \rangle) [x \leftarrow t'] & \xrightarrow{\circ} (u' C \langle t' \rangle) [x \leftarrow t'] \\
\equiv_{@r} & \\
u' C \langle x \rangle [x \leftarrow t'] & \dashv \xrightarrow{\circ} u' C \langle t' \rangle [x \leftarrow t']
\end{aligned}$$

- 3. Inductive case 1: inside an abstraction.** Suppose that $t = lx.t' \rightarrow lx.u' = u$. We consider two subcases, depending on whether the \Leftrightarrow step is internal to the body of the abstraction, or involves the outermost abstraction:
- If the application of the \Leftrightarrow step is internal to t' , we have by *i.h.*:

$$\begin{array}{ccc} t' & \xrightarrow{\quad} & u' \\ \equiv & & \equiv \\ w' & \dashrightarrow & r' \end{array}$$

so is immediate to conclude that:

$$\begin{array}{ccc} lx.t' & \xrightarrow{\quad} & lx.u' \\ \equiv & & \equiv \\ lx.w' & \dashrightarrow & lx.r' \end{array}$$

- If the outermost abstraction takes part in the \Leftrightarrow step, then a \equiv_λ step must have been applied, so t' must be of the form $t''[y \leftarrow u']$. We consider two further subcases, depending on whether the commuted substitution is involved in the reduction step:
 - If the reduction step $t''[y \leftarrow u'] \rightarrow w'$ is an exponential, and the commuted substitution $[y \leftarrow u']$ is the one contracted by the exponential step, then the situation is exactly like in case 2e (*Commutation with abstraction* for exponential steps), by reading the diagram from the bottom up.
 - Otherwise, note that there cannot be a multiplicative step at the root, and that the step cannot be internal to u' , as LO contexts do not go inside substitutions. Therefore the reduction step must be internal to t'' and the situation is:

$$\begin{array}{ccc} lx.t''[y \leftarrow u'] & \xrightarrow{\quad} & lx.u''[y \leftarrow u'] \\ \equiv_\lambda & & \equiv_\lambda \\ (lx.t'')[y \leftarrow u'] & \dashrightarrow & (lx.u'')[y \leftarrow u'] \end{array}$$

- 4. Inductive case 2: left of an application.** Suppose that $t = t' q \rightarrow u' q = u$. If the application of the \Leftrightarrow step is internal to t' , we may immediately conclude by *i.h.* (analogous to case 3a). The interesting case is when the outermost application takes part in the \Leftrightarrow step. There are two possibilities, depending on whether a $\equiv_{@1}$ step or a $\equiv_{@r}$ step is applied:
- $\equiv_{@1}$ step.** Then t' must be of the form $t''[x \leftarrow w']$. We consider two further subcases, depending on whether the commuted substitution is involved in the reduction step:
 - If the reduction step $t''[x \leftarrow w'] \rightarrow r'$ is an exponential step and the commuted substitution $[x \leftarrow w']$ is also the one contracted by the exponential step, then the situation is exactly like in case 2f (*Left commutation with application* for exponential steps), by reading the diagram from the bottom up.

ii. Otherwise, note that the reduction step cannot be internal to w' , since LO contexts do not go inside substitutions, so it must be internal to t'' and the situation is:

$$\begin{array}{ccc} t''[x \leftarrow w'] q & \xrightarrow{\quad} & u''[x \leftarrow w'] q \\ \equiv_{@1} & & \equiv_{@1} \\ (t'' q)[x \leftarrow w'] & \dashrightarrow & (u'' q)[x \leftarrow w'] \end{array}$$

(b) **$\equiv_{@r}$ step.** Then q must be of the form $q'[x \leftarrow w']$ and the situation is:

$$\begin{array}{ccc} t' q'[x \leftarrow w'] & \xrightarrow{\quad} & u' q'[x \leftarrow w'] \\ \equiv_{@r} & & \equiv_{@r} \\ (t' q')[x \leftarrow w'] & \dashrightarrow & (u' q')[x \leftarrow w'] \end{array}$$

5. **Inductive case 3: right of an application.** Suppose that $t = q t' \rightarrow q u' = u$. If the application of the \Leftrightarrow step is internal to t' , we may immediately conclude by *i.h.* (analogous to case 3a). The interesting case is when the outermost application takes part in the \Leftrightarrow step. There are two possibilities, depending on whether a $\equiv_{@1}$ step or a $\equiv_{@r}$ step is applied:

(a) **$\equiv_{@1}$ step.** Then q must be of the form $q'[x \leftarrow w']$ and the situation is:

$$\begin{array}{ccc} q'[x \leftarrow w'] t' & \xrightarrow{\quad} & q'[x \leftarrow w'] u' \\ \equiv_{@1} & & \equiv_{@1} \\ (q' t')[x \leftarrow w'] & \dashrightarrow & (q' u')[x \leftarrow w'] \end{array}$$

(b) **$\equiv_{@r}$ step.** Then t' must be of the form $t''[x \leftarrow w']$. We consider two further subcases, depending on whether the commuted substitution is involved in the reduction step:

- i. If the reduction step $t''[x \leftarrow w'] \rightarrow r'$ is an exponential step and the commuted substitution $[x \leftarrow w']$ is also the one contracted by the exponential step, then the situation is exactly like in case 2g (*Right commutation with application* for exponential steps), by reading the diagram from the bottom up.
- ii. Otherwise, note that the reduction step cannot be internal to w' , since LO contexts do not go inside substitutions, so it must be internal to t'' and the situation is:

$$\begin{array}{ccc} q t''[x \leftarrow w'] & \xrightarrow{\quad} & q u''[x \leftarrow w'] \\ \equiv_{@r} & & \equiv_{@r} \\ (q t'')[x \leftarrow w'] & \dashrightarrow & (q u'')[x \leftarrow w'] \end{array}$$

6. **Inductive case 4: left of a substitution.** Suppose that $t = t'[x \leftarrow q] \rightarrow u'[x \leftarrow q] = u$. If the application of the \Leftrightarrow step is internal to t' , we may immediately conclude by *i.h.* (analogous to case 3a). The interesting case is

when the outermost substitution node takes part in the \Leftrightarrow step. There are four possibilities, depending on whether a \equiv_{gc} step, a \equiv_{com} step, a $\equiv_{[.]}$ step, or a \equiv_{dup} step is applied:

- (a) **\equiv_{gc} step.** The reduction step cannot be internal to q , since LO contexts may not go inside substitutions, so the step must be internal to t' , and closing the diagram is trivial:

$$\begin{array}{ccc} t'[x \leftarrow q] & \xrightarrow{\quad} & u'[x \leftarrow q] \\ \equiv_{\text{gc}} & & \equiv_{\text{gc}} \\ t' & \xrightarrow{\quad} & u' \end{array}$$

Note that if $x \notin \text{fv}(t')$ then $x \notin \text{fv}(u')$ by the usual property that reduction does not create free variables.

- (b) **\equiv_{com} step.** Then t' must be of the form $t''[y \leftarrow w']$ with $x \notin \text{fv}(w')$. We consider two further subcases, depending on whether the commuted substitution is involved in the reduction step:

- i. If the reduction step $t''[y \leftarrow w'] \rightarrow r'$ is an exponential step and the commuted substitution $[y \leftarrow w']$ is also the one contracted by the exponential step, then the situation is exactly like in case 2b (*Commutation of independent substitutions* for exponential steps), by reading the diagram from the bottom up.
- ii. Otherwise, note that the reduction step cannot be internal to w' , since LO contexts may not go inside substitutions, so it must be internal to t'' , and the situation is:

$$\begin{array}{ccc} t''[y \leftarrow w'][x \leftarrow q] & \xrightarrow{\quad} & u''[y \leftarrow w'][x \leftarrow q] \\ \equiv_{\text{com}} & & \equiv_{\text{com}} \\ t''[x \leftarrow q][y \leftarrow w'] & \xrightarrow{\quad} & u''[x \leftarrow q][y \leftarrow w'] \end{array}$$

- (c) **$\equiv_{[.]}$ step.** Two cases, depending on whether the $\equiv_{[.]}$ step is applied from left to right or from right to left:

- i. $\equiv_{[.]}$ is applied from left to right. Then t' must be of the form $t''[y \leftarrow w']$ with $x \notin \text{fv}(t'')$. We consider two further subcases, depending on whether the commuted substitution is involved in the reduction step:

- A. If the reduction step $t''[y \leftarrow w'] \rightarrow r'$ is an exponential step and the commuted substitution $[y \leftarrow w']$ is also the one contracted by the exponential step, then the situation is exactly like in case 2(c)ii (*Composition of substitutions* for exponential steps), by reading the diagram from the bottom up.

- B. Otherwise, note that the reduction step cannot be internal to w' , since LO contexts may not go inside substitutions, so it must be internal to t'' , and the situation is:

$$\begin{array}{ccc} t''[y \leftarrow w'][x \leftarrow q] & \xrightarrow{\quad} & u''[y \leftarrow w'][x \leftarrow q] \\ \equiv_{[.]} & & \equiv_{[.]} \\ t''[y \leftarrow w'[x \leftarrow q]] & \xrightarrow{\quad} & u''[y \leftarrow w'[x \leftarrow q]] \end{array}$$

Note that if $x \notin \text{fv}(t'')$, then $x \notin \text{fv}(u'')$, by the usual fact that reduction does not create free variables.

- ii. $\equiv_{[]} \text{ is applied from right to left.}$ Then q must be of the form $q'[y \leftarrow w']$, and the reduction step must be internal to t' , so the situation is:

$$\begin{array}{ccc}
 t'[x \leftarrow q'[y \leftarrow w']] & \xrightarrow{\quad} & u'[x \leftarrow q'[y \leftarrow w']] \\
 \equiv[\cdot] & & \equiv[\cdot] \\
 t'[x \leftarrow q'][y \leftarrow w'] & \dashrightarrow & u'[x \leftarrow q'][y \leftarrow w']
 \end{array}$$

- (d) **\equiv_{dup} step.** Two cases, depending on whether the \equiv_{dup} step is applied from left to right or from right to left:

- i. \equiv_{dup} is applied from left to right. Then the reduction step is internal to t' and closing the diagram is immediate:

$$t'[x \leftarrow q] \xrightarrow{\equiv_{\text{dup}}} u'[x \leftarrow q]$$

$$t'_{[y]_x}[x \leftarrow q][y \leftarrow q] \dashv \vdash u'_{[y]_x}[x \leftarrow q][y \leftarrow q]$$

- ii. \equiv_{dup} is applied from right to left. Then t' must be of the form $t''[y \leftarrow q]$. We consider two further subcases, depending on whether the commuted substitution is involved in the reduction step:

- A. If the reduction step $t''[y \leftarrow q] \rightarrow r'$ is an exponential step and the affected substitution $[y \leftarrow q]$ is also the one contracted by the exponential step, then t'' must be of the form $C'_{[x]_y} \langle y \rangle$ and the situation is:

$$C'_{[x]_y} \langle y \rangle [y \leftarrow q][x \leftarrow q] \xrightarrow{\quad \mathsf{e} \quad} C'_{[x]_y} \langle q \rangle [y \leftarrow q][x \leftarrow q]$$

\equiv_{dup}

$$C' \langle y \rangle [y \leftarrow q] \dashdash \xrightarrow{\quad \mathsf{e} \quad} C' \langle q \rangle [y \leftarrow q]$$

\equiv_{dup}

- B. Otherwise, note that the reduction step cannot be internal to q , since LO contexts may not go inside substitutions, so it must be internal to t'' . The situation is then exactly like in case Lemma 6(d)i, by reading the diagram from the bottom up.

□